L Number	Hits	Search Text	DB	Time stamp
1	32	((((((resource near4 manag$5) (resource near4 allocat$3) (resource near4 partition$4))) and ((soft logical$4 software) near4 partition$4)) and ((address$3 near4 translat$3) and table and (os o/s (operating adj2 system)))) and ((modif$5 chang$3 edit$3 updat$3) same (address$3 near3 translat$3) same (access$5))) and (((physical near4 resource) i/o memory) same (allocat$3 reserv$5))	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/23 08:54
2	49	(((((((resource i/o memory) near4 manag$5) ((resource i/o memory) near4 allocat$3) ((resource i/o memory) near4 partition$4))) and ((soft logical$4 software user) near4 partition$4)) and ((address$3 near4 translat$3) and table and (os o/s (operating adj2 system)))) and ((modif$5 chang$3 edit$3 updat$3) same (address$3 near3 translat$3) same (access$5))) and ((((physical hardware) near4 resource) i/o memory) same (allocat$3 reserv$5))	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/23 09:37
3	17	(((((((resource i/o memory) near4 manag$5) ((resource i/o memory) near4 allocat$3) ((resource i/o memory) near4 partition$4))) and ((soft logical$4 software user) near4 partition$4)) and ((address$3 near4 translat$3) and table and (os o/s (operating adj2 system)))) and ((modif$5 chang$3 edit$3 updat$3) same (address$3 near3 translat$3) same (access$5))) and ((((physical hardware) near4 resource) i/o memory) same (allocat$3 reserv$5))) not (((((((resource near4 manag$5) (resource near4 allocat$3) (resource near4 partition$4))) and ((soft logical$4 software) near4 partition$4)) and ((address$3 near4 translat$3) and table and (os o/s (operating adj2 system)))) and ((modif$5 chang$3 edit$3 updat$3) same (address$3 near3 translat$3) same (access$5))) and (((physical near4 resource) i/o memory) same (allocat$3 reserv$5)))	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/23 08:59
4	32	(((((((resource i/o memory) near4 manag$5) ((resource i/o memory) near4 allocat$3) ((resource i/o memory) near4 partition$4))) and ((soft logical$4 software user) near4 partition$4)) and ((address$3 near4 translat$3) and table and (os o/s (operating adj2 system)))) and ((modif$5 chang$3 edit$3 updat$3) same (address$3 near3 translat$3) same (access$5))) and ((((physical hardware) near4 resource) i/o memory) same (allocat$3 reserv$5))) not ((((((((resource i/o memory) near4 manag$5) ((resource i/o memory) near4 allocat$3) ((resource i/o memory) near4 partition$4))) and ((soft logical$4 software user) near4 partition$4)) and ((address$3 near4 translat$3) and table and (os o/s (operating adj2 system)))) and ((modif$5 chang$3 edit$3 updat$3) same (address$3 near3 translat$3) same (access$5))) and ((((physical hardware) near4 resource) i/o memory) same (allocat$3 reserv$5))) not ((((((resource near4 manag$5) (resource near4 allocat$3) (resource near4 partition$4))) and ((soft logical$4 software) near4 partition$4)) and ((address$3 near4 translat$3) and table and (os o/s (operating adj2 system)))) and ((modif$5 chang$3 edit$3 updat$3) same (address$3 near3 translat$3) same (access$5))) and (((physical near4 resource) i/o memory) same (allocat$3 reserv$5))))	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/23 10:41
12	2039	((map$4 same (resource hardware physical i/o cpu processor memory) same (logical$2 partition$3) same address$3) and (os (operating adj2 system)) and (allocat$3 reserv$5))	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/23 13:31
13	181	((map$4 same (resource hardware physical i/o cpu processor memory) same logical$2 same partition$3 same address$3) and (os (operating adj2 system)) and (allocat$3 reserv$5))	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/23 13:32
14	98	((map$4 same (resource hardware physical i/o cpu processor memory) same logical$2 same partition$3 same address$3 same access$5) and (os (operating adj2 system)) and (allocat$3 reserv$5))	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/23 13:34

16	42	(((map$4 same (resource hardware physical i/o cpu processor memory) same logical$2 same partition$3 same address$3 same access$5) and (os (operating adj2 system)) and (allocat$3 reserv$5))) and (((request$3 access$5) near5 (den$3 refus$3 grant$3 permi$5 allow$3 authori$6 error$3 exception)) and table and translat$3 and address$3)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2003/09/23 13:39

US006212614B1

(12) **United States Patent**
Hoerig et al.

(10) **Patent No.: US 6,212,614 B1**
(45) **Date of Patent: Apr. 3, 2001**

(54) **LEGACY MIL-STD-1750A SOFTWARE EMULATOR ADDRESS TRANSLATION USING POWER PC MEMORY MANAGEMENT HARDWARE**

(75) Inventors: **Timothy R. Hoerig**, Beavercreek; **William J. Cannon**, Centerville, both of OH (US)

(73) Assignee: **TRW Inc.**, Redondo Beach, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/451,431**

(22) Filed: **Nov. 30, 1999**

Related U.S. Application Data

(63) Continuation-in-part of application No. 09/002,960, filed on Jan. 5, 1998, now Pat. No. 6,041,402.

(51) **Int. Cl.**[7] **G06F 12/06**
(52) **U.S. Cl.** **711/209**; 711/202
(58) **Field of Search** 711/202, 206, 711/208, 209

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,276,594	*	6/1981	Morley	713/600
4,760,525	*	7/1988	Webb	712/2
5,079,737	*	1/1992	Hackbarth	711/164
5,548,746	*	8/1996	Carpenter	710/3
5,822,749	*	10/1998	Agarwal	707/2
5,956,752	*	9/1999	Mathews	711/204

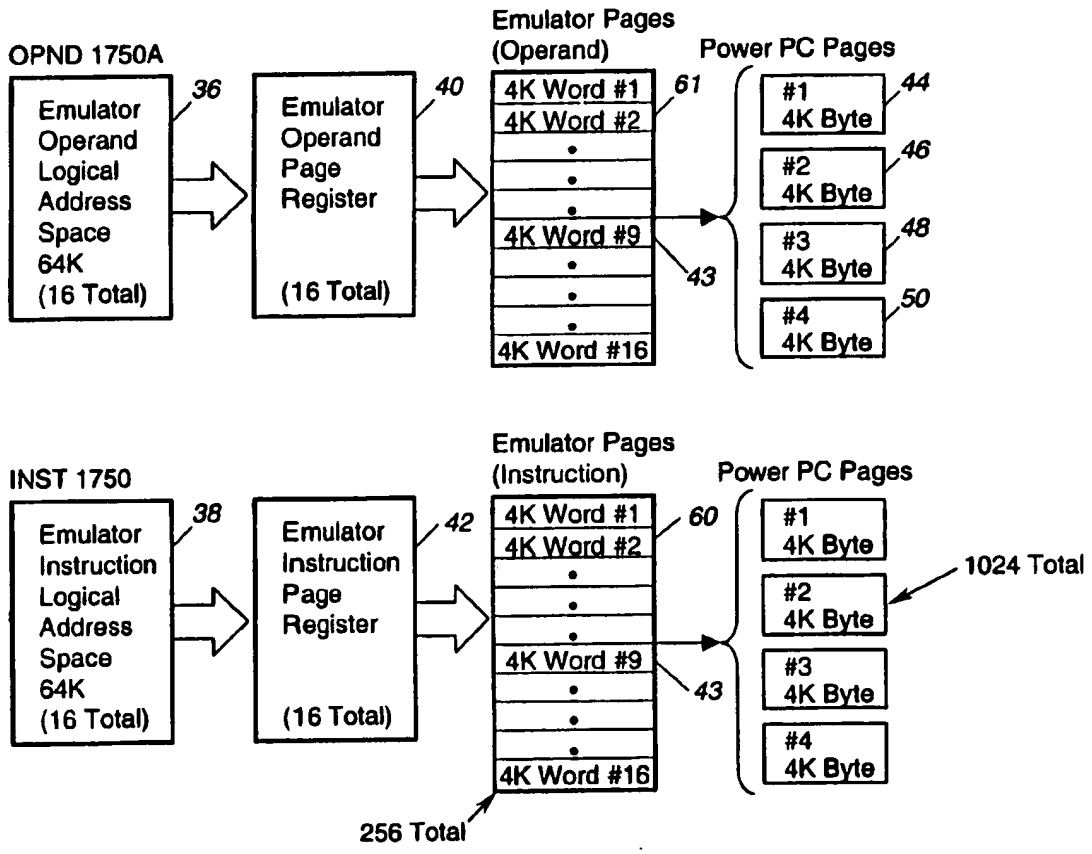
* cited by examiner

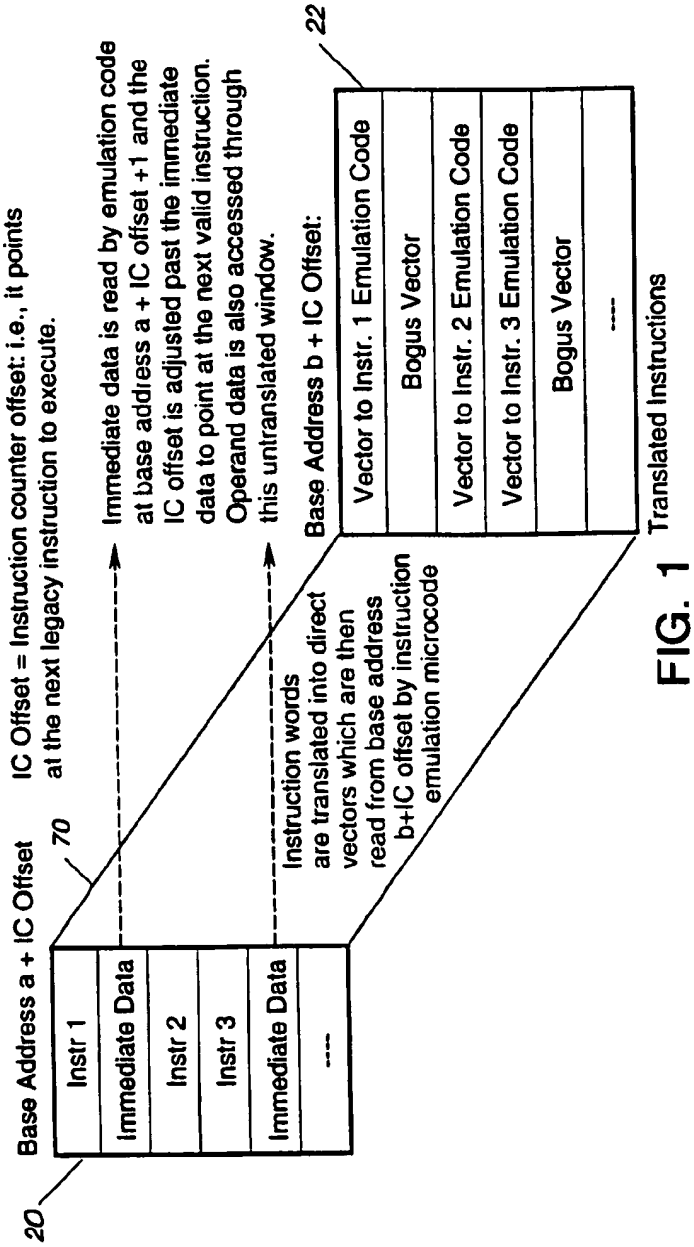
Primary Examiner—Eric Coleman
(74) *Attorney, Agent, or Firm*—Michael S. Yatsko

(57) **ABSTRACT**

A system and method for implementing the paging and protection attributes, such as block protection and access lock and key functions promulgated in MIL-STD-1750A. The present invention takes advantage of the PowerPC microprocessor architecture to implement the paging and protection attributes required by MIL-STD-1750A in hardware. Since the paging and the protection attributes are implemented in hardware, the system performance is greatly improved.

11 Claims, 13 Drawing Sheets




Base Address a + IC Offset
IC Offset = Instruction counter offset: i.e., it points at the next legacy instruction to execute.
Immediate data is read by emulation code at base address a + IC offset +1 and the IC offset is adjusted past the immediate data to point at the next valid instruction. Operand data is also accessed through this untranslated window.
Instruction words are translated into direct vectors which are then read from base address b+IC offset by instruction emulation microcode
Base Address b + IC Offset:
Translated Instructions
Instr 1
Immediate Data
Instr 2
Instr 3
Immediate Data

Vector to Instr. 1 Emulation Code
Bogus Vector
Vector to Instr. 2 Emulation Code
Vector to Instr. 3 Emulation Code
Bogus Vector

20
70
22

FIG. 1

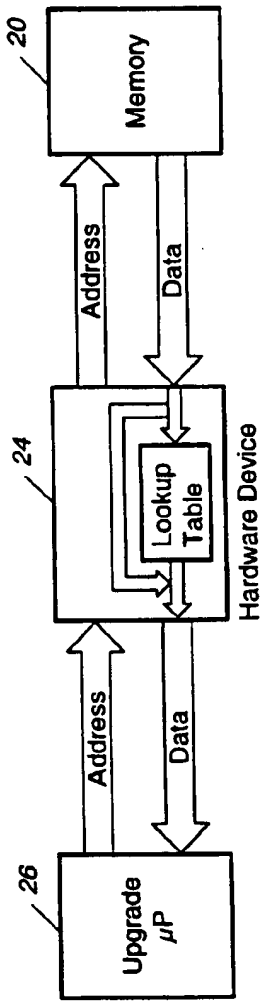

20
Memory
Address
Data
24
Lookup Table
Hardware Device
Address
Data
26
Upgrade µP

FIG. 2

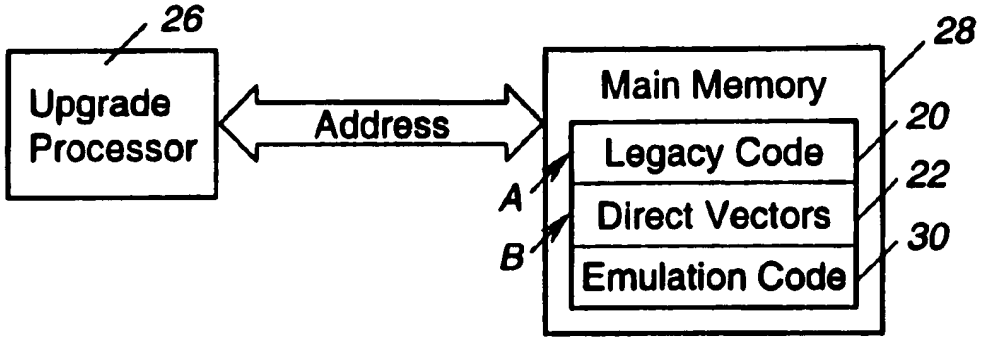

26
Upgrade Processor
Address
28
Main Memory
20
Legacy Code
22
Direct Vectors
30
Emulation Code
A
B

FIG. 3

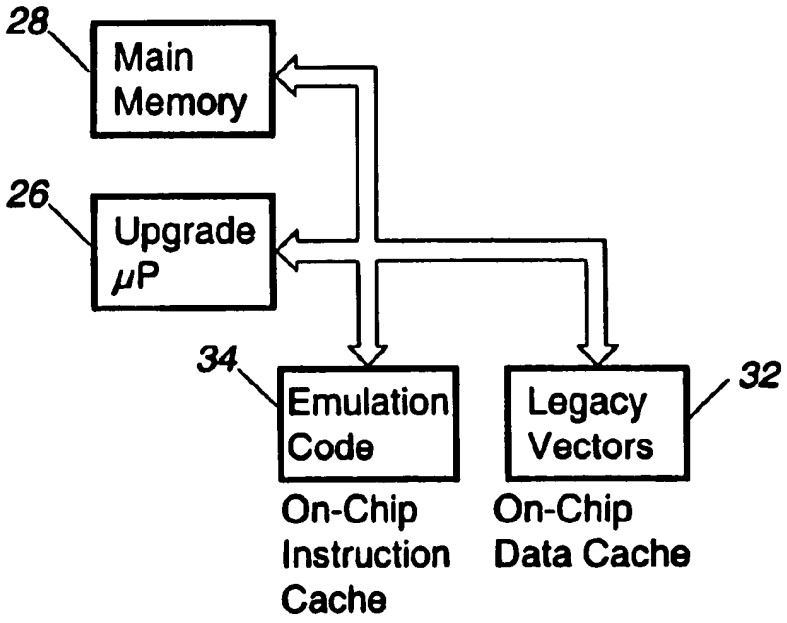

28
Main Memory
26
Upgrade μP
34
Emulation Code
32
Legacy Vectors
On-Chip Instruction Cache
On-Chip Data Cache

FIG. 4

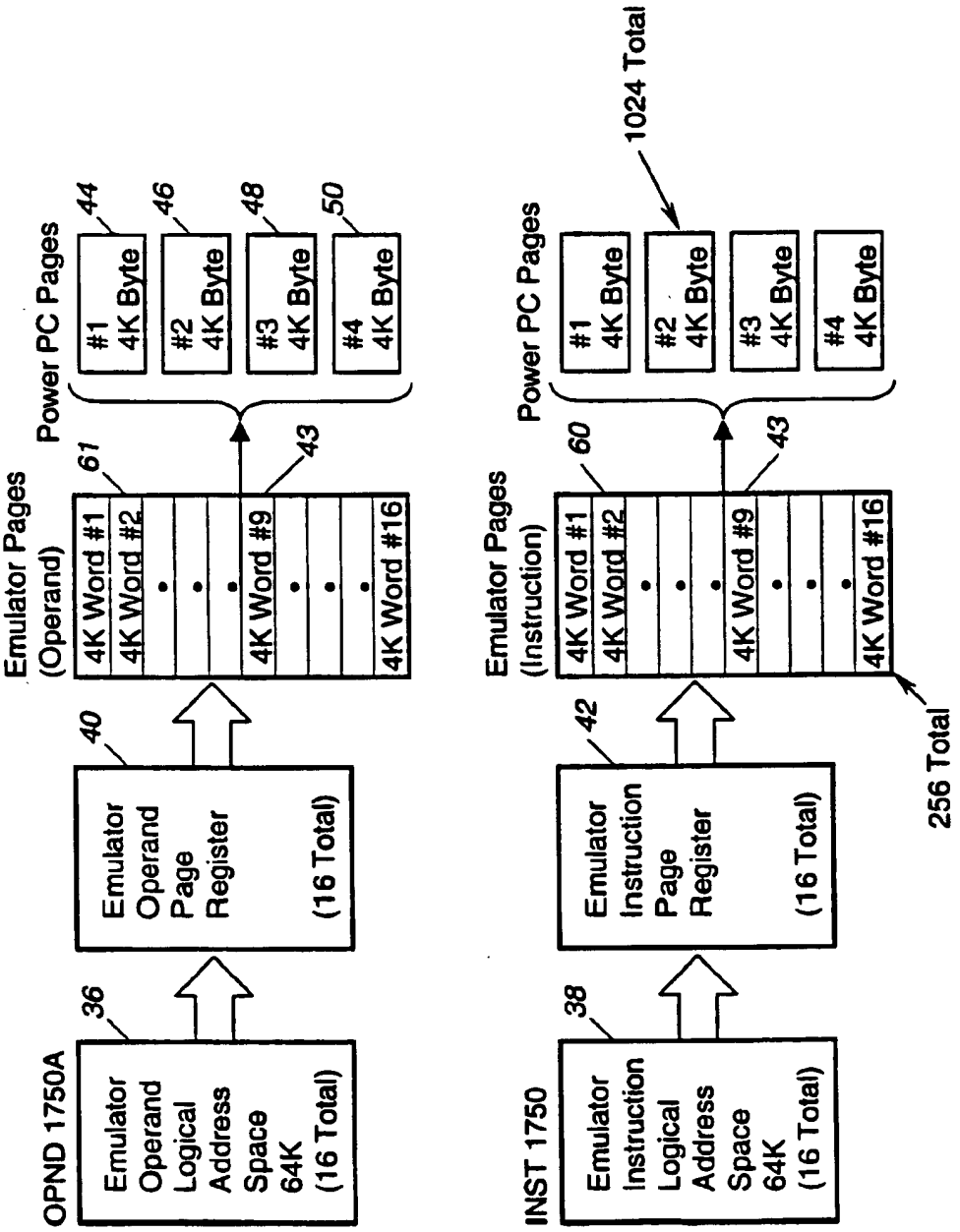

OPND 1750A
Emulator Operand Logical Address Space 64K (16 Total)
36
Emulator Operand Page Register (16 Total)
40
Emulator Pages (Operand)
61
4K Word #1
4K Word #2
4K Word #9
4K Word #16
43
Power PC Pages
44
#1 4K Byte
46
#2 4K Byte
48
#3 4K Byte
50
#4 4K Byte
INST 1750
Emulator Instruction Logical Address Space 64K (16 Total)
38
Emulator Instruction Page Register (16 Total)
42
Emulator Pages (Instruction)
60
4K Word #1
4K Word #2
4K Word #9
4K Word #16
43
Power PC Pages
1024 Total
#1 4K Byte
#2 4K Byte
#3 4K Byte
#4 4K Byte
256 Total

FIG. 5

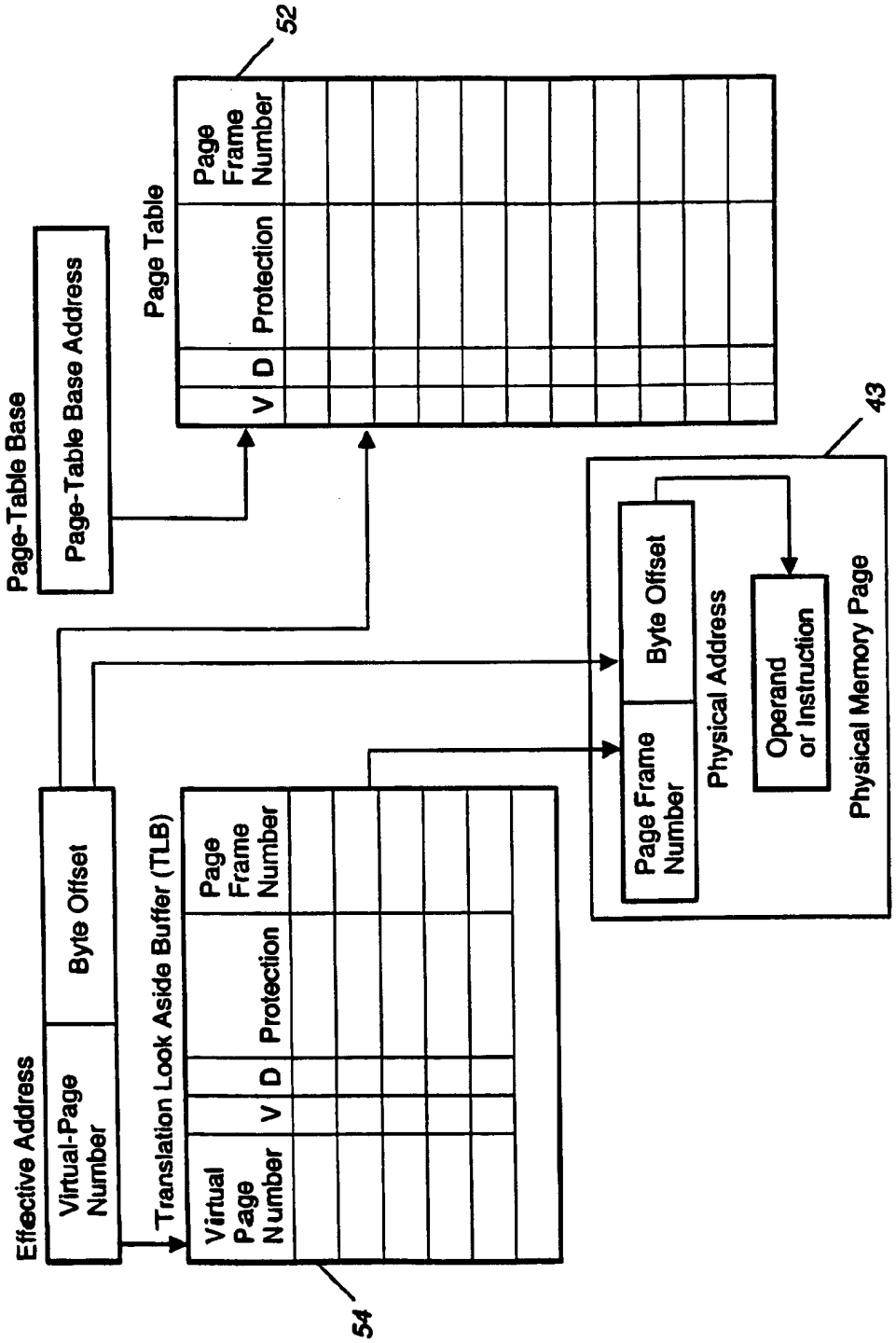

Page-Table Base
Page-Table Base Address
Page Table
52
Page Frame Number
Protection
D
V
Effective Address
Virtual-Page Number
Byte Offset
Translation Look Aside Buffer (TLB)
Virtual Page Number
54
Physical Address
Page Frame Number
Byte Offset
Operand or Instruction
Physical Memory Page
43

FIG. 6

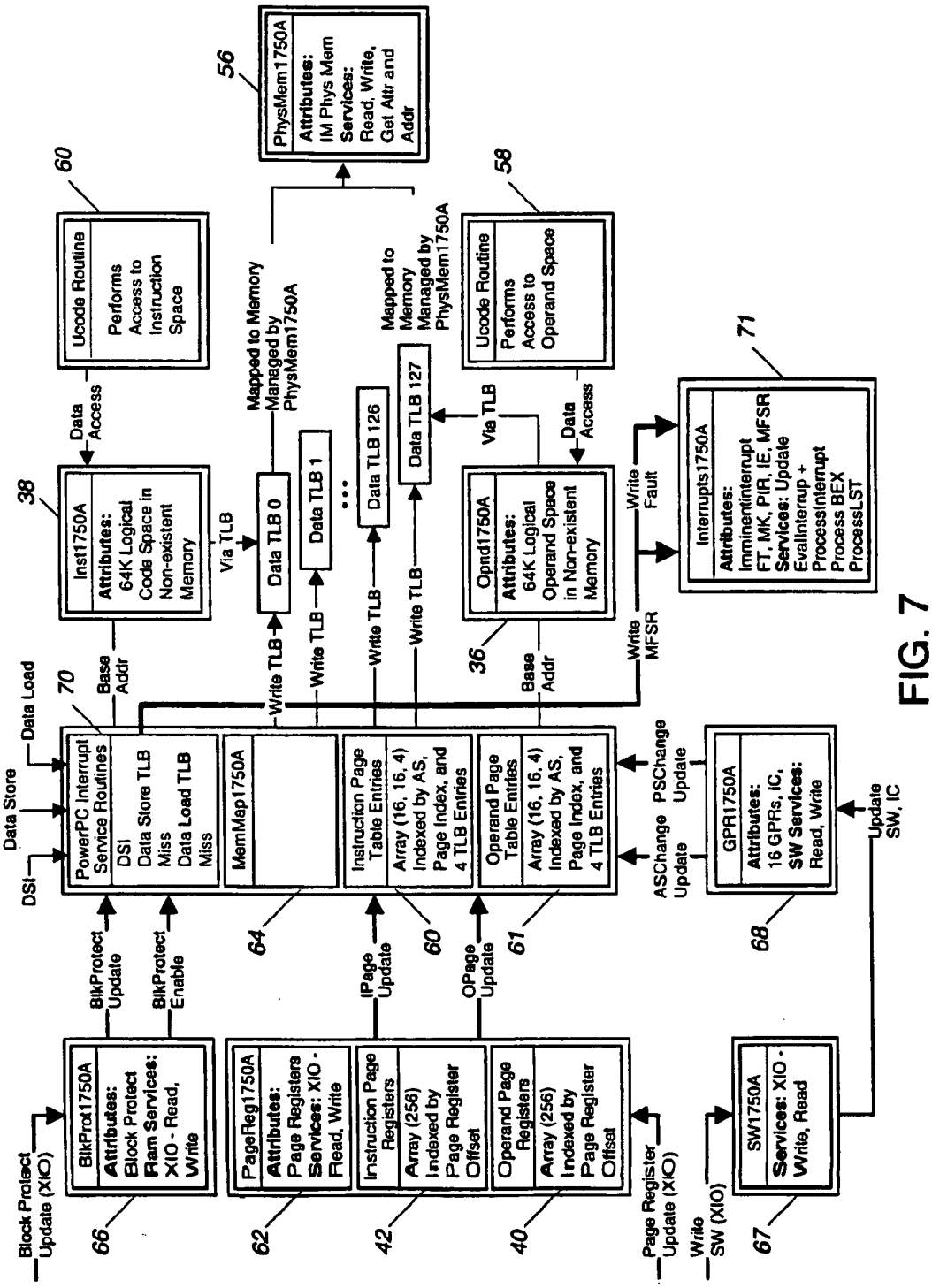

PhysMem1750A
Attributes:
IM Phys Mem
Services:
Read, Write,
Get Attr and
Addr
56
Ucode Routine
Performs
Access to
Instruction
Space
60
Ucode Routine
Performs
Access to
Operand Space
58
Inst1750A
Attributes:
64K Logical
Code Space in
Non-existent
Memory
38
Opnd1750A
Attributes:
64K Logical
Operand Space
in Non-existant
Memory
36
Interrupts1750A
Attributes:
ImminentInterrupt
FT, MK, PIR, IE, MFSR
Services: Update
EvalInterrup +
ProcessInterrupt
Process BEX
ProcessLST
71
Data Access
Via TLB
Mapped to Memory
Managed by
PhysMem1750A
Data TLB 0
Data TLB 1
Data TLB 126
Data TLB 127
Write TLB
Write Fault
Write MFSR
Base Addr
Data Load
Data Store
DSI
PowerPC Interrupt
Service Routines
DSI
Data Store TLB
Miss
Data Load TLB
Miss
70
MemMap1750A
64
Instruction Page
Table Entries
Array (16, 16, 4)
Indexed by AS,
Page Index, and
4 TLB Entries
60
Operand Page
Table Entries
Array (16, 16, 4)
Indexed by AS,
Page Index, and
4 TLB Entries
61
GPR1750A
Attributes:
16 GPRs, IC,
SW Services:
Read, Write
68
ASChange Update
PSChange Update
Update SW, IC
BlkProtect Update
BlkProtect Enable
IPage Update
OPage Update
BlkProt1750A
Attributes:
Block Protect
Ram Services:
XIO - Read,
Write
66
PageReg1750A
Attributes:
Page Registers
Services: XIO -
Read, Write
62
Instruction Page
Registers
Array (256)
Indexed by
Page Register
Offset
42
Operand Page
Registers
Array (256)
Indexed by
Page Register
Offset
40
SW1750A
Services: XIO -
Write, Read
67
Block Protect
Update (XIO)
Page Register
Update (XIO)
Write
SW (XIO)

FIG. 7

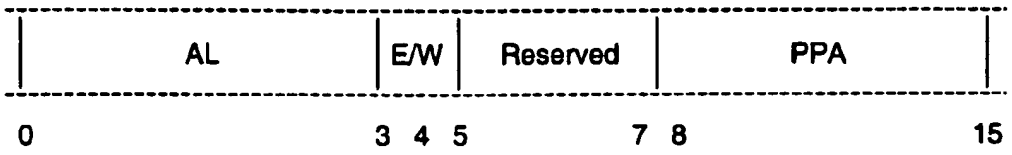


FIG. 8

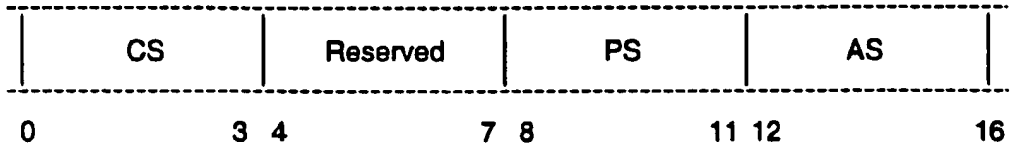


FIG. 9

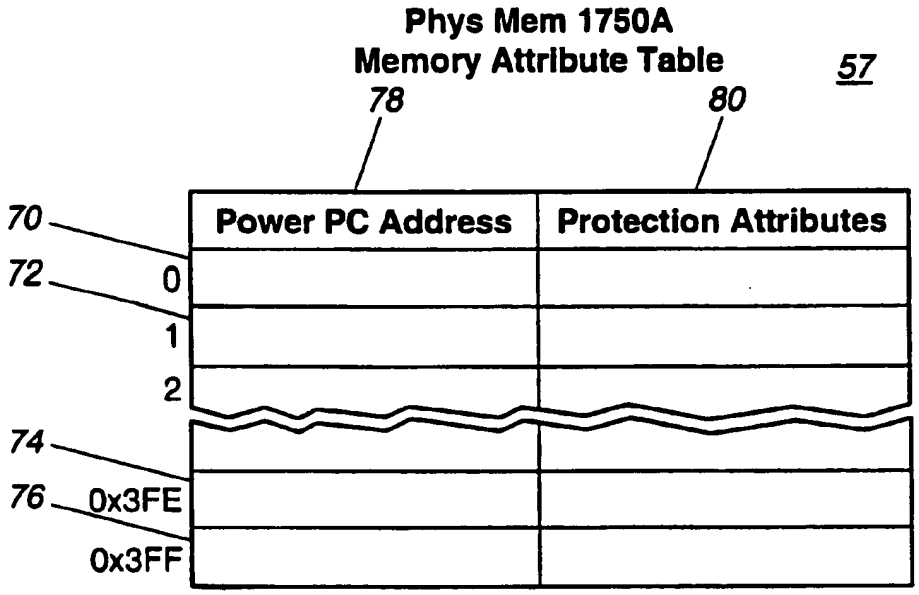

Phys Mem 1750A
Memory Attribute Table
57
78
80
70
72
74
76
Power PC Address
Protection Attributes
0
1
2
0x3FE
0x3FF

FIG. 10

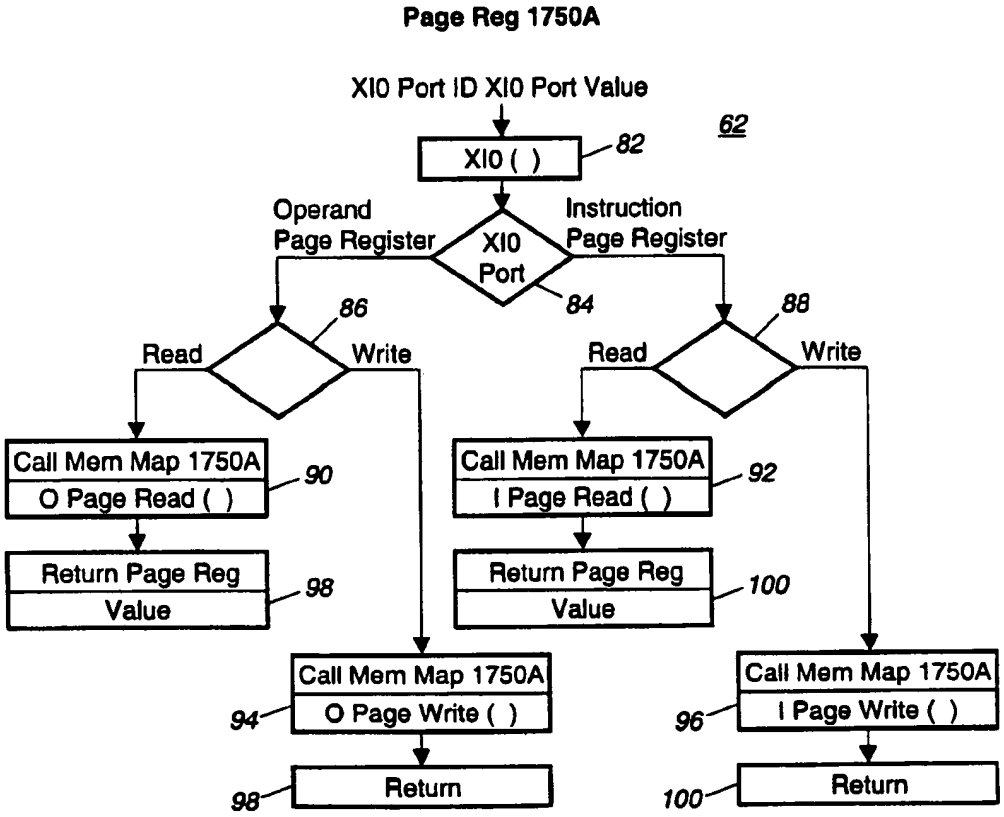

Page Reg 1750A
XIO Port ID XIO Port Value
62
XIO ( )
82
Operand Page Register
XIO Port
Instruction Page Register
84
86
88
Read
Write
Read
Write
Call Mem Map 1750A
O Page Read ( )
90
Call Mem Map 1750A
I Page Read ( )
92
Return Page Reg Value
98
Return Page Reg Value
100
Call Mem Map 1750A
O Page Write ( )
94
Call Mem Map 1750A
I Page Write ( )
96
Return
98
Return
100

FIG. 11

Mem Map 1750A

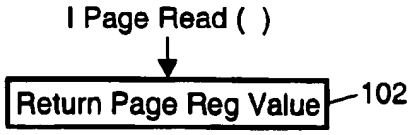


FIG. 12A

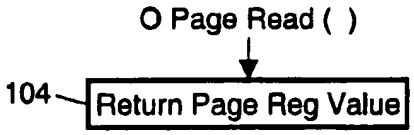


FIG. 12B

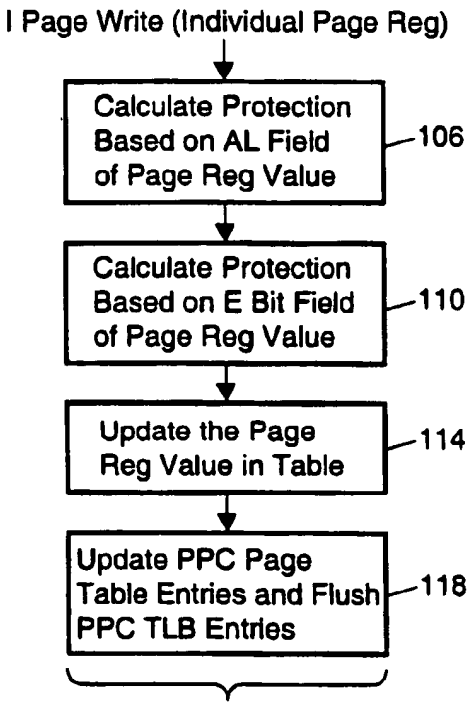


FIG. 12C

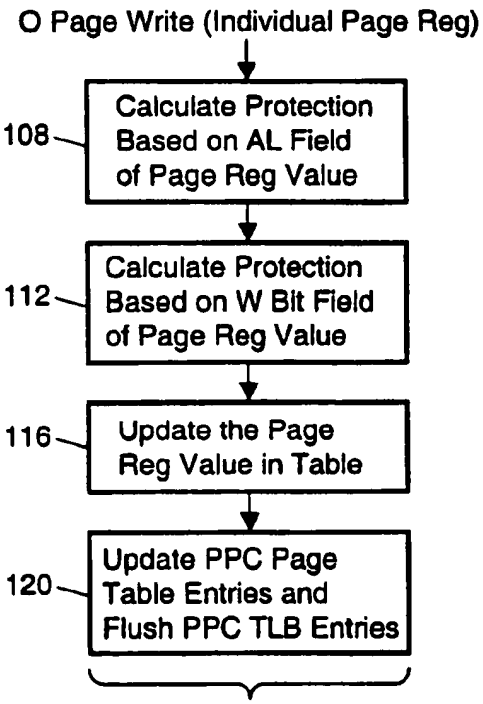


FIG. 12D

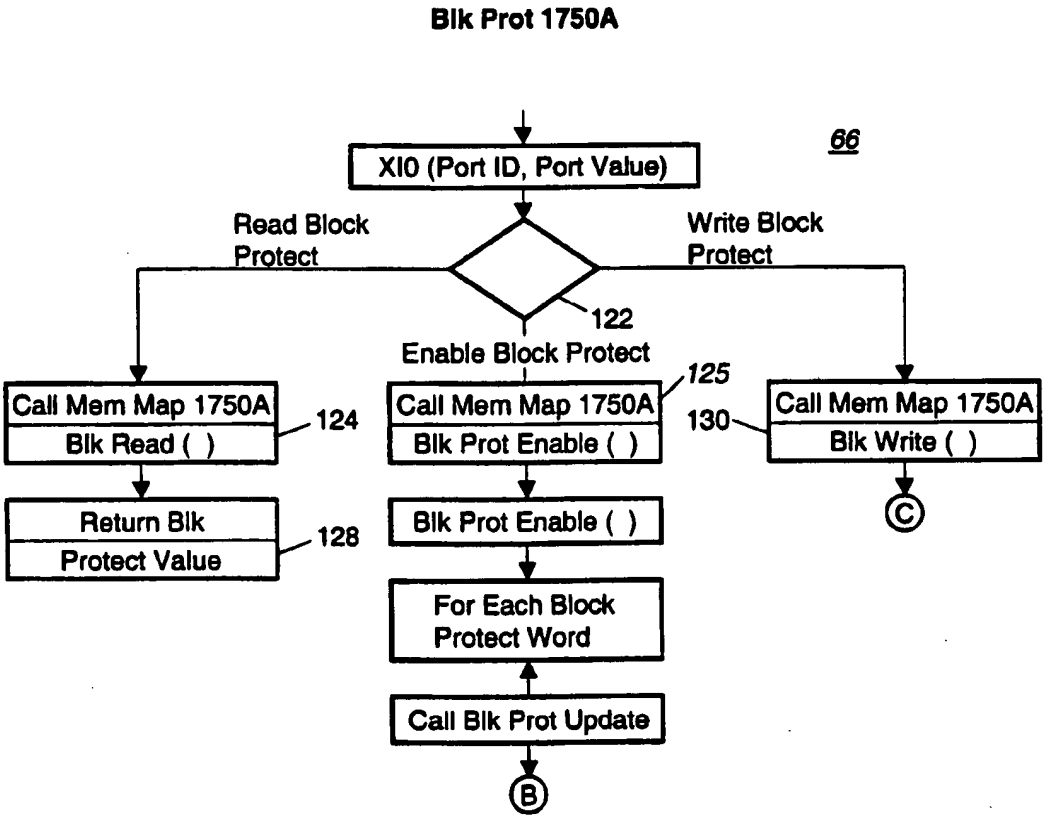

Blk Prot 1750A
66
XIO (Port ID, Port Value)
Read Block Protect
Write Block Protect
122
Enable Block Protect
125
Call Mem Map 1750A
Blk Read ( )
124
Call Mem Map 1750A
Blk Prot Enable ( )
130
Call Mem Map 1750A
Blk Write ( )
Return Blk Protect Value
128
Blk Prot Enable ( )
For Each Block Protect Word
Call Blk Prot Update
B
C

FIG. 13A

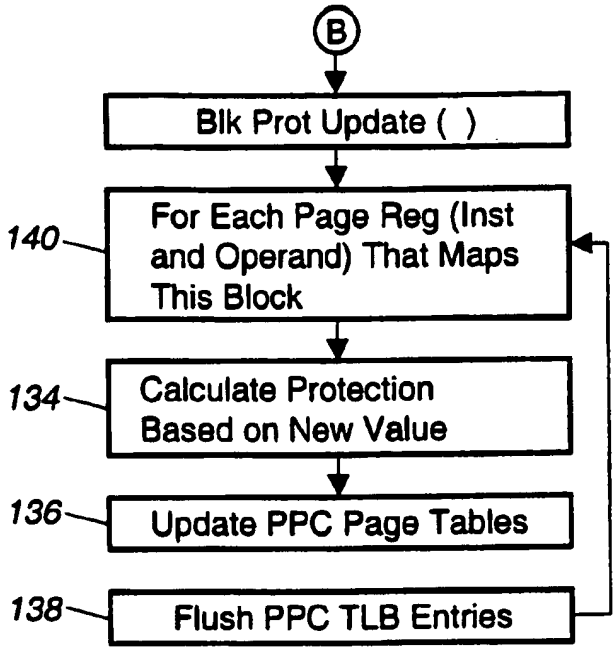


FIG. 13B

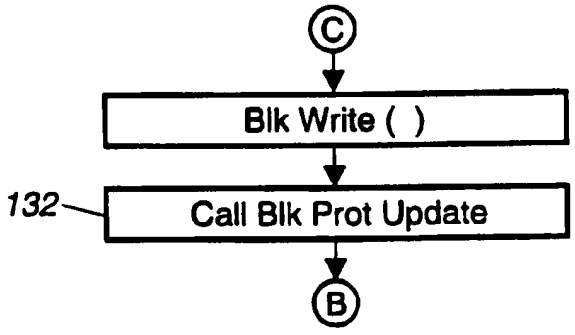


FIG. 13C

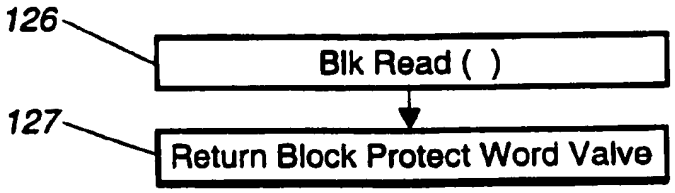


FIG. 13D

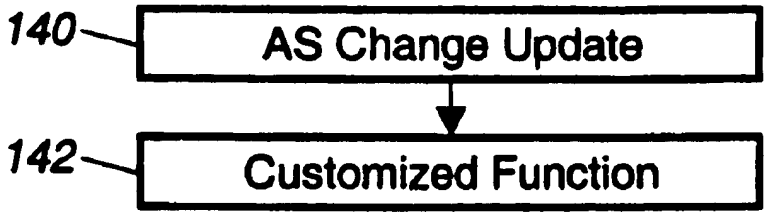


FIG. 14A

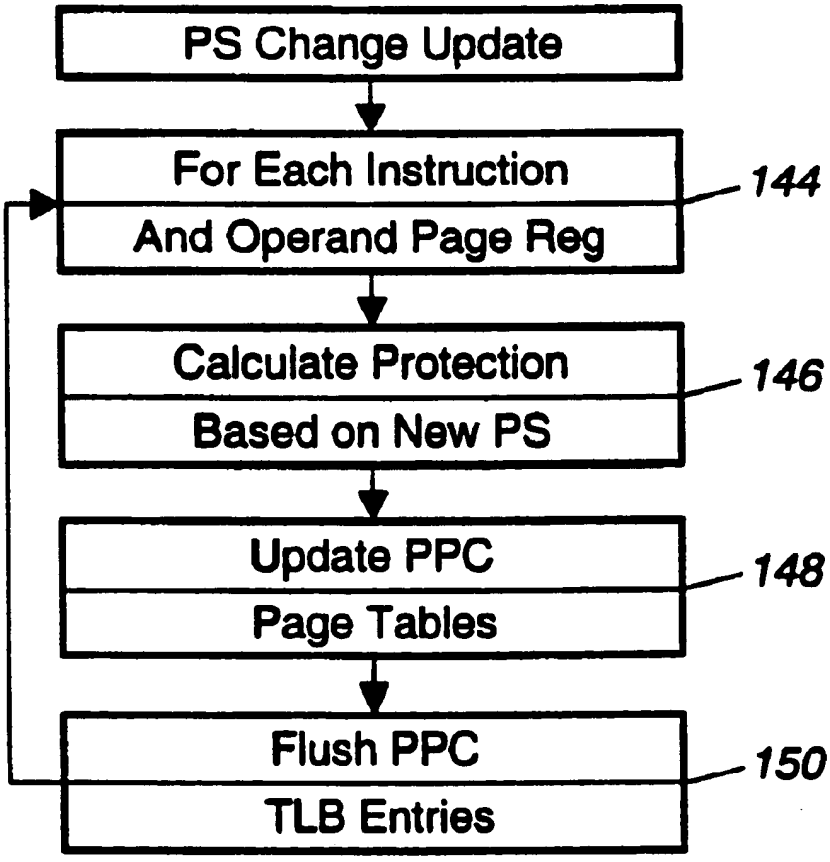


FIG. 14B

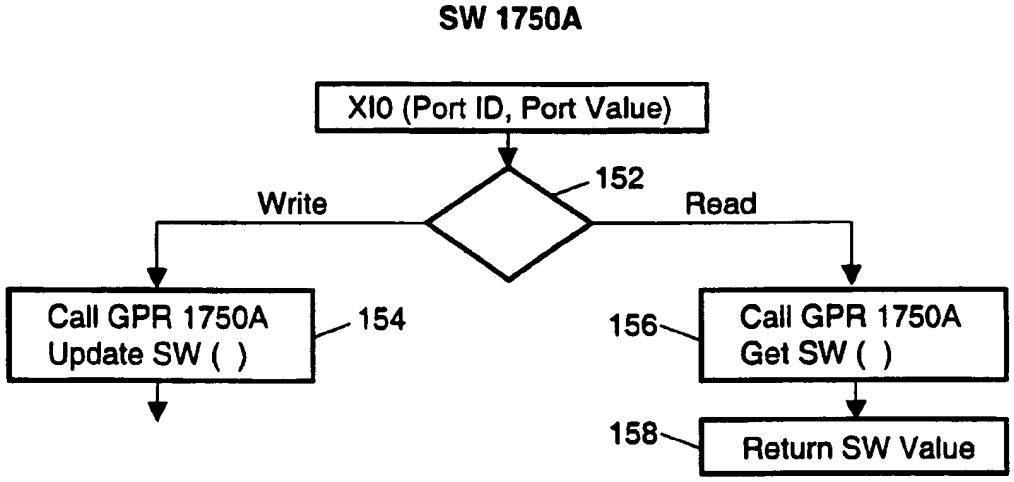


FIG. 15A

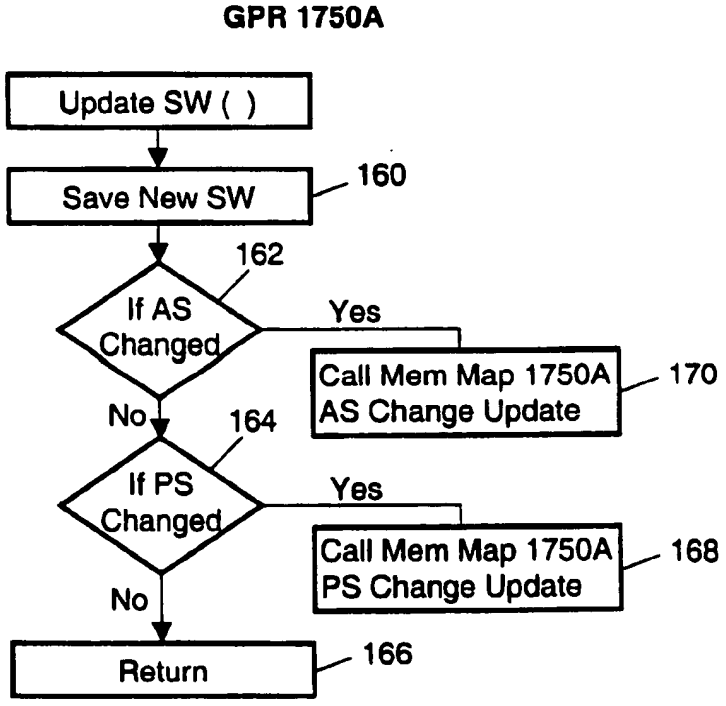


FIG. 15B

LEGACY MIL-STD-1750A SOFTWARE EMULATOR ADDRESS TRANSLATION USING POWER PC MEMORY MANAGEMENT HARDWARE

CROSS REFERENCE TO RELATED APPLICATIONS

This application is a continuation-in-part of commonly owned application Ser. No. 09/002,960, now U.S. Pat. No. 6,041,402, filed on Jan. 5, 1998. This application is also related to commonly owned copending patent application, filed on even date, entitled; "Concurrent Legacy and Native Code Execution Techniques"; by T. Hoerig, et al., application Ser. No. 09/451,156.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to a memory management system and more particularly to memory management system which can be used with a software emulator which meets MIL-STD-1750A with respect to paging and protection attributes, such as block protection and access lock and key functions.

2. Description of the Prior Art

It is known that microprocessors are configured with different instruction set architectures (ISA). The ISA determines the instruction set for a particular microprocessor. Application programs are executed by the microprocessors normally written in relatively high level language, which is compiled into machine instructions compatible with the instruction set for the specific microprocessor. Microprocessors are increasingly being designed to execute instructions faster and faster. As such, systems incorporating such microprocessors are often upgraded to increase the speed of the system. Unfortunately, depending on the particular upgrade, often times the instruction set of the upgrade microprocessor is incompatible with the instruction set of the microprocessor to be replaced ("legacy microprocessor"). As such, in such applications, the existing application programs often need to be rewritten in new and modem computer languages with modem compilers. Unfortunately, such an undertaking can be quite cumbersome and expensive.

Due to the age and obsolescence of many existing avionic onboard computers, the reliability of such computers is rapidly declining while maintenance is becoming more difficult and costly to achieve. As such, it is sometimes required to replace outdated "legacy" microprocessors with newer technology "native" microprocessors. To work around instructions set incompatibilities, emulation systems (emulators) have been developed. Emulators are known which emulate the instructions set of the legacy microprocessor in order to enable the instructions of the legacy microprocessor to be "executed" by a different microprocessor. Both software and hardware based emulators are known. For example, various software emulators for the F-16 avionics integration support facility (AISF) common modular environment (COMET) are described in document no. F-16AISF-COMET-100 (EMULATORS-SWD-A, dated May 21, 1996). Hardware based emulators for military standard MIL-STD-1750A, are discussed in the document entitled Line Replaceable Unit Emulator Hardware Product Fabrication Specification, document no. SFF20702 dated Apr. 16, 1996.

Unfortunately, known software emulators have been known to be relatively inefficient. In particular, in such known software emulators, legacy instructions are fetched or the upgrade microprocessor which uses a look up table to interpret the legacy instruction. Since each legacy instruction must be interpreted, computer systems which incorporate cache memory are known to suffer from relatively high probability of cache misses which decreases the overall throughput of the system.

Another problem with such software emulators is the need to comply with MIL-STD-1750A with respect to memory management. As used herein MIL-STD-1750A refers to "Military Standard Sixteen Bit Computer Instruction Set Architecture", dated Jul. 2, 1980, hereby incorporated by reference. MIL-STD-1750A sets forth a virtual paging and protection scheme that maps sixteen (16) 64K word logical operand spaces and sixteen (16) 64K word logical instruction spaces into 1 M word of physical memory. Each 64K word logical space is controlled by sixteen (16) logical page registers which control the logical to physical mapping as well as the protection attributes, such as execute protect and write protect, as well as an access lock and key functions. Block protection is specified for physical memory in blocks of 1K word size. Block protection over-rides page register protection. In other words, if the block is write protected, it is write protected regardless of the write protection setting in the page registers.

To accurately model the protection attributes in a software emulator, each operand access and each instruction access require software to look up the current settings for the page register and determine the protection. If the attribute enables access, then the access is performed, otherwise a fault is generated. Such protection attributes are known to seriously degrade the performance of such software emulators. Thus, there is a need to efficiently implement paging and block protect, and lock and key functions which meet MIL-STD-1750A for a software emulator.

SUMMARY OF THE INVENTION

Briefly the present invention relates to a system and method for implementing the paging and protection attributes, such as block protection and access lock and key functions promulgated in MIL-STD-1750A. The present invention takes advantage of the PowerPC microprocessor architecture to implement the paging and protection attributes required by MIL-STD-1750A in hardware. Since the paging and the protection attributes are implemented in hardware, the system performance is greatly improved.

BRIEF DESCRIPTION OF THE DRAWING

These and other objects of the present invention will be readily understood with reference to the following specification and attached drawing, wherein:

FIG. **1** is a block diagram illustrating the translation of the legacy instruction in accordance with the present invention.

FIG. **2** is a block diagram illustrating one embodiment (in which vector translation is done by hardware) for translating the legacy instructions in accordance with the present invention.

FIG. **3** is a block diagram illustrating the mapping of the legacy instructions to main memory.

FIG. **4** is a block diagram of an upgrade microprocessor with cache memory for illustrating the operation of the cache system in accordance with the present invention.

FIG. **5** is a diagram of the memory mapping in accordance with the present invention.

FIG. **6** is a diagram illustrating the virtual paging architecture of a PowerPC microprocessor.

FIG. **7** is an exemplary block diagram of the system in accordance with the present invention.

FIG. **8** is a bit map of the word format used in logical page registers in accordance with MIL-STD-1750A.

FIG. **9** is a bit map of a status word register in accordance with MIL-STD-1750A.

FIG. **10** is a block diagram of the PhysMem 1750A function block illustrated in FIG. **7**.

FIG. **11** is a filter for the PageReg 1750A function block illustrated in FIG. **11**.

FIGS. **12A–12D** are flowcharts representative of the MemMap 1750A function block illustrated in FIG. **7**.

FIG. **13A** is a block diagram for BlkProt 1750A function block illustrated in FIG. **7**.

FIG. **13B** is a block diagram for BlkProt Update function in accordance with the present invention.

FIG. **13C** is a block diagram for a Blk Write function in accordance with the present invention.

FIG. **13D** is a block diagram of a Blk Read function in accordance with the present invention.

FIG. **14A** is a block diagram of an ASchange Update function in accordance with the present invention.

FIG. **14B** is a block diagram for a PSchange Update function in accordance with the present invention.

FIG. **15A** is a block diagram of a SW 1750A function block illustrated in FIG. **7**.

FIG. **15B** is a block diagram of an Update SW function in accordance with the present invention.

DETAILED DESCRIPTION OF THE INVENTION

The present invention relates to a emulation system which meets MIL-STD-1750A. FIGS. **1–4** relate to a software emulation system and method for emulating legacy instructions of an outdated ("legacy") microprocessor with a new upgrade microprocessor with an incompatible instruction set FIGS. **5–9** relate to a method for implementing the paging and protection attributes, such as block protection and access lock and key functions features specified in MIL-STD-1750A.

DIRECT VECTORED LEGACY INSTRUCTION SET EMULATION

In applications in which a new upgrade "native" microprocessor replaces an outdated "legacy microprocessor with a different instruction set, software programs are written which emulate each of the legacy instructions using instructions of the upgrade microprocessor. In known emulation systems, the emulation software causes the legacy instruction to be fetched and interpreted by the new microprocessor by way of the lookup table. As discussed in more detail below, such methodology has a significant impact on the throughput of the system. In order to increase the throughput of the system, the legacy instructions are translated into direct vectors to software routines or emulation code for emulating the legacy instructions. As such, as will be discussed in more detail below, the probability of cache misses is greatly reduced which results in increased throughput of the system.

Turning to FIG. **1**, embedded software or code for a legacy microprocessor is generally identified with the reference numeral **20**. Such code is normally stored in non-volatile read only memory (ROM). As shown, the ROM **20** includes legacy instructions, identified as INSTR **1**, INSTR **2** and INSTR **3**, etc. The ROM **20** also includes immediate data. The instructions INSTR **1**, INSTR **2**, INSTR **3**, etc. plus the immediate data in the ROM **20**, are located, for example, at a base address A within the memory space.

In accordance with an important aspect of the invention, each instruction (i.e. instruction INSTR **1**, INSTR **2**, INSTR **3**, etc.) is translated to a direct vector to a software routine or emulation code for emulating the legacy instruction. For example, each legacy instruction is mapped or translated to another memory device **22**, such as a ROM **22**. The structure of the memory device **22** with respect to the instructions, is maintained the same as the ROM **20** but at a different base address B. In other words, instructions in the ROM **20** are located at a base address A plus an instruction counter (IC) offset which corresponds to the next legacy instruction to be executed. The instructions, INSTR1, INSTR **2**, INSTR3, etc. are mapped to the ROM **22** at a different base address B but with the same IC offset.

The direct vectors in the ROM **22** can either be JUMP instructions to software routines for emulating the legacy instruction or address pointers. For example, the direct vectors can represent an offset pointer to an emulation microcode routine or a pointer to a table which contains a pointer to a microcode routine. Regardless, it is these vectors that are fetched by the emulation software rather than the legacy instructions.

The immediate data may be translated into bogus vectors in the ROM **22** which are not used by the emulation software. Rather, the emulation software in accordance with present invention may access the immediate data directly from the legacy code **20** by reading the data directly at the base address A from the ROM **20**.

Various methods are suitable for translating the legacy microcode. Both hardware and software methods for translating these instructions are suitable. FIG. **2** illustrates a hardware implementation in which a hardware device **24**, such as a ROM containing a lookup table, is coupled to the data bus between an upgrade microprocessor **26** and the legacy microcode, i.e. ROM **20**. The hardware device **24** is configured such that at any time an access to the legacy code is requested, i.e. base address B plus IC offset, the vector corresponding to the requested instruction is provided. Alternately, the decoder can be bypassed such that an access to the legacy code (i.e. base address A plus IC offset) will return the untranslated data. Thus, the upgrade processor can be directed to the associated emulation code routine by the fetched vector, or it can access immediate data directly from memory.

In an alternate embodiment of the invention, the legacy emulation code may be translated by software when the legacy memory is loaded into main memory or modified. In particular, a software program, for example, a portion of the initialization software, is used to load the legacy code, into the computer system main memory **28** (FIG. **3**) (e.g. at base address A). In this implementation, after loading the legacy microcode, the direct vectors (i.e. ROM **22**) are loaded into the main memory **28** at another location (e.g. base address B), allowing the emulation code in the upgrade processor to access either the translated **22** or untranslated **20** legacy memory sections. The vectors retrieved from the translated memory **22** are used to point to the individual software routines in the emulation code **30**. With this embodiment of the invention, the translated **22** and untranslated **20** legacy memory sections need not be disjoint, they might be interleaved, such that each vector immediately follows or proceeds the untranslated instruction.

As mentioned above, the configuration of the method and system for emulating legacy instructions is adapted to improve the throughput of the system. More particularly, many known microprocessors include cache memories in order to improve the throughput of the system. Software fetched from the main memory, is copied into the cache memory, which is much quicker than main memory. Thus, instructions stored in the cache memory can be executed much quicker than those stored only in main memory. Such cache memories are normally formed from high speed static random access memory (SRAM) and are used to store copies of data in the main memory or newly stored data. Such cache memories operate on the principles that most programs execute instructions in sequence, and, due to loops, programs are likely to re-use recently fetched instructions. This principle is call locality. Thus, instead of fetching a single instruction at a time, a cache memory system looks ahead and fetches blocks of instructions in sequence and stores the instructions for quick access.

In operation, all data stored in a cache memory is stored with what is known as an address tag. The address tag indicates the physical addresses of the data in the main memory that is being stored in the cache. Whenever the microprocessor initiates a memory access, the address tags in the cache memory are first examined to determine if the particular data requested is already stored in the cache memory. When the data is found in the cache memory, this is known as a cache hit and data is immediately available to the microprocessor. If it is determined that the requested data is not in the cache memory, this condition is known as a cache miss. As a result of a cache miss, the requested data then must be retrieved from the main memory at a much slower rate.

FIG. **4** illustrates a typical configuration of a microprocessor with onboard cache memory. In previous known systems, the known emulation software fetched the legacy instructions themselves. The legacy instructions were then interpreted by way of lookup table. Since the cache memory is based on the premise that a sequence of data will be requested in sequential memory locations, the use of the lookup table is not particularly efficient in cache memory systems and results in a relatively high probability of cache misses. By utilizing direct vectors, and because of effects of the locality principle in the legacy code and corresponding direct vectors, the probability of cache misses is greatly reduced thereby increasing the overall throughput of the system. More particularly, referring to FIG. **4**, a memory system in accordance with the present invention is illustrated. As shown, the system includes the upgrade microprocessor **26** which includes two onboard cache memories **32** and **34**. One cache memory **32** is used for data, forming a data cache while the other cache memory **34** is used for instructions forming an instruction cache. The instruction cache **34** may be used almost exclusively for the emulation microcode. The data cache **32** may be used for the legacy code.

In operation, a group of eight vectors may be fetched from main memory upon a cache miss and stored in the data cache **32** as part of the cache line refill operation. Since legacy instructions normally proceed in sequential order, the subsequent 7 requests for instruction vectors will normally be resident in the data cache **30**. If the next legacy instructions to be executed is non-sequential but is within the last one thousand instructions to executed, (i.e. local loops), there is a high probability that the vector will still be in the data cache **30**. This invention has reduced the probability of cache misses and thus increased the throughput of the system.

LEGACY MIL-STD-1750A SOFTWARE

Emulator Address Translation

Microprocessors and software used in airborne applications are required to comply with MIL-STD-1750A. As discussed above, oftentimes outdated "legacy" microprocessors are replaced with newer "native" microprocessors. Since the native microprocessors are used with legacy software, software emulators have been developed to emulate the instructions of the legacy microprocessor. These native microprocessors as well as the software emulators need to be in compliance with the aforementioned standard. One aspect of the aforementioned standard relates to memory paging with associated protection attributes, such as block protection and access lock and key functions. In particular, MIL-STD-1750A applies to 16 bit computer systems and requires separate logical address space for instructions and operands. In particular, MIL-STD-1750A requires sixteen (16) 64K word logical operand or data spaces and sixteen (16) 64K word logical instruction spaces to be mapped into 1 M word of physical memory. Each 64K word logical space is controlled by sixteen (16) page registers which control the logical to physical mapping as well as the protection attributes, such as execute protect and write protect as well as an access lock and key. In addition, the MIL-STD-1750A specifies block protection for physical memory in blocks of 1K word sizes. As discussed above, meeting the requirements of MIL-STD-1750A can cause serious performance degradation in software emulators.

The system in accordance with the present invention utilizes the virtual paging architecture, available on microprocessors constructed in accordance with the PowerPC architecture, as generally shown in FIG. **6**, to implement the paging and protection attributes specified in MIL-STD-1750A. Since the paging and protection attributes are implemented in hardware, emulation of the paging with protection attributes in accordance with MIL-STD-1750A operates more efficiently than other known systems.

As discussed above, the MIL-STD-1750A specifies that the memory space be configured into sixteen (16) 64K word logical operand spaces (OPND 1750A) and (16) 64K word logical instruction spaces (INST 1750A). Referring to FIG. **5**, only one of the 16 operand logical address spaces **36** and one instruction logical address spaces **38** is illustrated for convenience. Referring to the operand logical address space, each of the operand logical address spaces **36** is mapped by sixteen (16) hardware logical page registers **40** into sixteen (16) emulator logical address space pages; each page generally identified with the reference numeral **43** and configured as 4K word on 4K word boundaries. Thus, the sixteen (16) emulator operand logical address 64K word spaces **36** are mapped into 256 4 K word emulator pages **43** for a total of 1 M word of address space. The mapping of the emulator instruction logical address **38** is similar as illustrated in FIG. **5** and is mapped into the same 1 M word of address space.

In accordance with an important aspect of the invention, the paging requirements set forth in MIL-STD-1750A can easily be implemented by utilizing a microprocessor conforming to the PowerPC architecture. In order to take advantage of the PowerPC architecture with respect to paging as well as the protection attributes, the system in accordance with the present invention represents each word in the emulator pages **43** as 4 bytes to the Power PC environment. As such, this allows one (1) emulator operand page **43** to be mapped to four (4) PowerPC pages **44,46,48** and **50** of physical memory as shown in FIG. **5**. Thus, each

emulator operand 64K word logical address space **36** is mapped to sixteen (16) emulator logical 4K word address pages **43**, which, in turn, are mapped to four (4) PowerPC pages; each power PC page being 4K bytes. The sixteen (16) emulator operand 64K word logical address spaces will thus map to 1024 PowerPC 4K byte pages **36**. As such, one (1) Power PC page will map to a 1K word block of the emulator instruction logical address space **38**. By so configuring the PowerPC microprocessor, one (1) PowerPC page (4K bytes) will map to a 1K word block of the emulator operand logical address space. Since the PowerPC architecture provides for block protection of each PowerPC page, which, in turn, is mapped to 1K word of emulator operand logical address space **36**, block protection down to a granularity of 1K word as required by MIL-STD-1750 is satisfied.

Implementation of the virtual paging scheme by way of a microprocessor configured to the PowerPC architecture can be done rather conveniently and easily. Detailed information for programming a microprocessor conforming to the PowerPC architecture is set forth in "PowerPC" Microprocessor Family: The Programmers Reference Guide, document No. MPCPRG/D, dated October, 1995, available from Motorola, hereby incorporated by reference.

The virtual paging architecture of the PowerPC microprocessor is best understood with reference to FIG. **6** and as generally described in "Computer Architecture", by Robert J. Baron and Lee Higbie, published by Addison-Wesley Publishing Company, copyright 1992, pages 206–208, hereby incorporated by reference. In general, logical addresses are divided into two parts, a virtual page number and a word offset within a page. The division is done by partitioning the bits of the logical address, in this case, the logical addresses of the emulator operand and instruction pages **36** and **38**. Normally, the higher order bits are used for the page number while the lower order bits represent the offset. In order to account for the translation between logical and physical address locations within the memory, the operating system of the microprocessor normally utilizes a page table **52** in main memory. Such a page table **52** normally includes control information for each memory page. An exemplary page table **52** is illustrated in FIG. **6** and includes a column for a validity bit (V), which indicates whether the page is in memory; a dirty bit (D) which indicates whether the program has modified the page; and a protection field which indicates which users may access the page and how (for example read only or read and write) and a page frame number for the page.

Each row in the page table **52** represents a 4K byte page of memory in the PowerPC architecture environment. As discussed above, since each page in the PowerPC architecture environment is used to represent 1K word of legacy memory, block protection in 1K word blocks is easily accomplished, since the PowerPC architecture provides for block protection for each PowerPC page. As such, the block protection at the 1K word level of granularity for the emulator logical address space as required by the MIL-STD-1750A is greatly simplified and implemented simply by the block protection provided by the PowerPC microprocessor architecture.

As will be discussed below, the PowerPC architecture also includes a translation look aside buffer (TLB) **54** (FIG. **6**). The TLB **54** is an address translation cache which essentially has the same information as the page table **52** as generally shown in FIG. **6** to reduce accesses to the main memory and speed up the overall throughput of the system.

An exemplary block diagram for implementing the virtual paging and block protection requirements specified by MIL-STD-1750A is illustrated in FIG. **7**. As shown in FIG. **7**, blocks which include the suffix -1750A are part of the software emulator.

Referring to FIG. **7**, 1 M word of physical memory is represented by a PhyMem 1750A function block, identified with the reference numeral **56**. In particular, the PhysMem 1750A block **56** represents 1 M word of physical memory which may have additional attributes, for example; NOMEMORY indicating physical memory not implemented; ROM indicating physical memory as read only; RAM indicating physical memory as read write and NON-CACHEABLE indicating that the physical memory is off-board and should not be cached by the PowerPC microprocessor. The PhysMem 1750A function block **56** manages the physical memory in 1K word blocks.

A diagram of the memory attribute table **57** contained in the PhysMem 1750A function block **56** is illustrated in FIG. **10**. The memory attribute table **57** is divided into 1024 entries, such as the entries **70**, **72**, **74** and **76**. Each entry **70**, **72**, **74**, and **76** consists of a PowerPC address field, generally identified with the reference numeral **78**, and a protection attribute field **80**. As discussed above, 4K bytes in the PowerPC address field **78** are mapped into 1K words in order to provide protection attributes down to a 1K word block level. Thus, each entry in the PowerPC address field corresponds to 4K bytes. As discussed above, by mapping each word in the emulator environment to 4 bytes in the PowerPC environment, protection can be provided down to the 1K word level in order to comply with MIL-STD-1750A

As discussed above, the logical address space is divided up into (16) operand logical address spaces **36** and sixteen (16) instruction logical address spaces **38**. These operand logical address spaces **36** and the instruction logical address spaces **38** are each formed from sixteen (16) 64K word logical address spaces to provide 1 M word logical address space for use for operands and instructions. The sixteen (16) logical operand spaces **36** and the sixteen (16) instruction spaces **38** are presented as windows in the PowerPC memory space. All accesses to the operand **36** and instruction spaces **38** by the software emulator are by way of micro code routines, identified with the reference numerals **58** and **60** respectively. As discussed above, each of the sixteen (16) 64K word operand logical address spaces **36** and instruction logical address spaces **38** are mapped by logical page registers **40** and **42**, discussed above; each hardware register **40**, **42** mapping a 64K word address space to sixteen (16) 4K word pages **43** on 4K word boundaries as generally shown in FIG. **5**.

A bit map of the word format for the logical page registers **40**, **42** in accordance with MIL-STD-1750 is illustrated in FIG. **8**. As shown, the field AL (bits **0–3**) relates to an optional access lock and key feature. If the access lock and key feature is not used, bits **0–3** are reset to zero. If the access lock and key feature is used, bits **0–3** define the access lock code for the associated page register, used with the access key codes, discussed below, to determine access permission. Permissible values of the access lock codes and access key codes are provided in Table VII of the MIL-STD-1750A. Bit **4** is a dual functional bit; the function of which is determined by whether it is used with an instruction page register or an operand page register. For instruction page registers, bit **4** is defined as an E bit and determines the criteria for reading for instruction fetches. When E=1, any attempted instruction read designating the page register is denied access and a fault is generated. For operand page registers, bit **4** is defined as a W bit and used to determine the criteria for permitting write references to an operand

page register. When W=1, attempted writes to the associated page register are not permitted and a fault is generated. Bits 5–7 are reserved and are all set to 0. The PPA filed (bits 8–15) is used to define the physical page address.

The Power PC architecture includes a page table, such as the page table **52** illustrated in FIG. **6**, for mapping the logical address space to physical address space. More specifically, an instruction page table **60** is provided to map the emulator instruction logical address space **38** to physical memory. Similarly, a Power PC operand page table **61** is used for mapping the operand logical address space **36** to physical memory. The logical addresses in the operand logical address space **36** and the instruction logical address space **38** are partitioned as shown in FIG. **6** with the higher order bits in the logical address space corresponding to page frame numbers in the page table **52** and the lower order bits representing the byte offset for the address of the operand or instruction within that particular memory page **43**. Such page tables are normally maintained in main memory. In order to speed up memory accesses, the PowerPC architecture also keeps an internal cache of the most recently used page table entries, called a translation look aside buffer (TLB). In particular, whenever the PowerPC microprocessor performs a memory access, it compares the logical address of the access against entries in the TLB **54** and if it finds a match it uses the physical address in the TLB **54** to perform the access. If the required logical address is not in the TLB **54**, the PowerPC operating system searches the page table **52** looking for a match. If a match is found it is loaded into the TLB **54** and the access is completed. If the required entry is not found in a page table **52**, a page fault is then generated.

Referring to FIG. **7**, all updates to the software emulator page registers **40** and **42** are through specialized input/output instructions (XIO's) that map one XIO location to one page register. These XIO's are intercepted by a PageReg 1750A function block **62** and passes them to a MemMap 1750A function block **64**.

A flow chart for the PageReg 1750A function block **62** is illustrated in FIG. **11**. As discussed above, the PageReg 1750A function block **62** intercepts the XIO's in step **82** and determines in step **84** whether the XIO's are for an operand or instruction page register by examining the port address of the XIO. As discussed above, the MIL-STD 1750A requires separate logical operand and logical instruction spaces. Once the system determines whether the XIO is for an operand or instruction page register, the system then determines in step **86** or **88** whether the XIO is a read or write. After determining whether the XIO is an operand or an instruction page register access and whether it is a read or write access of that page register, the system calls the MemMap 1750A function block **64**, as indicated in steps **90**, **92**, **94** and **96**. As will be discussed in more detail below, the MemMap 1750A function block **64** merely returns the page register value in steps **98** and **100** for read operations. For write operations, as will be discussed in more detail below, the PowerPC page tables are manipulated and the protection attributes are calculated and returned in steps **98** and **100**.

Referring to FIG. **7**, when a data access is performed by the software emulator microcode (Ucode) routine **58** or **60** to the operand or instruction logical address space **36** or **38**, the PowerPC TLB **54** (FIG. **6**) is searched to determine if it contains an entry with the virtual page number of the PowerPC effective address of the operand or instruction data access **36** or **38**. If the TLB **54** does contain an entry with the virtual page number of the access, the PowerPC effective address is automatically translated to a corresponding PowerPC physical address according to FIG. **6** and the protection attributes of that entry are checked. If no entry in the TLB **54** contains the virtual page number of the access, the MemMap 1750A function block **64** updates the PowerPC page tables with the required synthesized page table entries and restarts the PowerPC instruction. If the protection attribute check indicates that the access being performed fails to meet the page table entry protection requirements, a PowerPC protection fault is generated. The MemMap 1750A function block **64** intercepts the PowerPC protection faults resulting from the attempted PowerPC access that did not meet the page table entry protection requirements. If the logical address of the fault is associated with an operand or instruction in the logical address spaces **36** or **38**, an exception is generated and passed back to the software emulator.

A flow chart for the MemMap 1750A function block **64** is illustrated in FIGS. **12A–12D**. As mentioned above, the PageReg 1750A function block **64** determines whether the XIO is for an instruction or an operand page register and whether the XIO is a write or read operation. For instruction and operand read operations, as illustrated in FIGS. **12A** and **12B**, the page register value is returned as indicated in steps **102** and **104**. As discussed above, the PowerPC tables include the protection attributes for each entry in the memory attribute table **57**. As such, the protection attributes are returned in steps **102** and **104**.

For instruction and operand page register write operations, the PowerPC page tables are manipulated and the protection attributes are calculated as indicated in FIGS. **12C** and **12D**. In particular referring to FIGS. **12C** and **12D**, for both instruction and operand page register write operations, the protection attributes are calculated in steps **106** and **108**, depending on the value in the AL field (FIG. **8**) of the logical page register **40** or **42**. As discussed above, the AL field relates to an optional access lock and key feature. If the access lock and key feature is not used, this field is reset to zero. Subsequently, in steps **110** and **112** the E/W field from the page register **40** or **42** is read and the protection is calculated based in the value of the E/W field. As discussed above, the instruction page registers define an E bit which determines the criteria for reading instruction fetches. For operand page registers, this bit is defined as a W bit and is used to determine the criteria for permitting writes to an operand page register. Subsequently, in steps **114**, **116**, **118** and **120** after the protection is determined based on the AL field and E/W field, the PowerPC page table entries are updated. In addition, the TLB **54** entries are flushed.

Block protection XIO's are intercepted by a BlkProt 1750A function block **66** and passed to the MemMap 1750A function block **64**. All updates to the software emulator block protects occur through the specialized XIO's that map one XIO location to sixteen (16) block protect entries (1 bit per block). These block protection XIO's are passed to the Power PC page table **52** (FIG. **6**) to specify write protection for each physical block. As discussed above, the Power PC architecture provides for write protection for each Power PC page. Since one Power PC 4K byte page corresponds to 1K word of logical address space, block protection is provided down to 1K word granularity level as required by MIL-STD-1750A.

A flow chart for the BlkProt 1750A function block **66** is illustrated in FIGS. **13A–13D**. Referring to FIG. **13A**, initially the system determines whether the XIO is a read operation or a write operation or an enable function for the enabling the block protect feature in step **122**. In steps **124**, **125** and **130**, the system calls the MemMap 1750A function block **64**. In particular, for read operations the system calls

MemMap 1750A and executes a BlkRead operation in step **126** as illustrated in FIG. **13**D. As shown in FIG. **13**D, the result of a BlkRead operation is a return of the block protection value in step **127**, which is subsequently returned in step **128** (FIG. **13**A) to complete the XIO read operation. For write operations, the system MemMap 1750A executes a BlkWrite operation in step **130**. As shown in FIG. **13**C, the BlkWrite operation makes a call to a BlkProt Update function as indicated in FIG. **13**B in step **132**. The BlkProt Update function calculates the value of the protection attribute based on the new value in step **134** for each page register entry that maps into the particular block. In step **136**, the PowerPC pages are updated. In step **138** the PowerPC affected TLB entries are flushed.

The selection of the currently active set of sixteen (16) instruction and sixteen (16) operand page registers **42** and **40** is controlled by the value of the AS field of the 1750A status word (SW) emulated in the GPR1750A function block **68**, as set forth in MIL-STD-1750A. All explicit program reads and writes of the SW are made through special XIOs which call the SW1750A function block **67**.

A bit map for the status word register is illustrated in FIG. **9** as specified in MIL-STD-1750A. The status word register is a 16 bit register and includes a condition status (CS) field (bits **0–3**), defined in the MIL-STD-1750A. Bits **4–7** of the status word register are reserved. Bits **8–11** relate to a processor state (PS) field. This field defines the access key for use with the access lock and key protection attribute. This PS field defines the memory access key code for all instruction and operand references to memory. The PS field also determines the criteria for execution of privileged instructions. When PS=0, privileged instructions are allowed to be executed. When PS≠0, privileged instructions are aborted and a fault is generated. Bits **12–15** relate to an address state (AS) field. The AS field essentially defines the particular one of 16 logical address spaces **36** or **38** that is requesting access. The access key codes are read by the Power PC and used to calculate the protection for the protection field in the page table **52** (FIG. **6**).

A block diagram for the SW 1750A function block **67** is illustrated in FIG. **15**A. As shown, the system receives XIO's and determines in step **152** whether the XIO is a write or a read operation. For write operations, the system calls the function blocks GPR 1750A in step **154** and executes an update SW instruction as discussed below. For read operations the system also calls the function block GPR 1750A in step **156** and executes a Get SW function and returns a status word value in step **158**. The GetSW function simply returns the current status word.

FIG. **15**B is a simplified flow chart for an Update SW instruction. In response to a request to write a new status word, the new status word is saved in step **160**. In step **162**, the system determines whether the address state AS field has changed in the page registers **40**, **42**. If not, the system proceeds to step **164** and determines whether the processor state PS field has changed. If not, the system returns in step **166**. If the PS field has changed, the system calls the function block MemMap 1750A in step **168** and executes a call to the PS change update, illustrated in FIG. **14**B, in step **168**. If the AS field has changed as determined in step **162**, the system makes a call to the function block MemMap 1750A in step **170** and executes an AS change update as indicated in FIG. **14**A.

Flow charts for updating the AS and PS fields are illustrated in FIG. **14**A and FIG. **14**B. Updates to the AS field are indicated with the reference numeral **140**. The changes to the AS field can be used for any number of customized functions as indicated by block **142**.

Changes to the PS field are indicated in FIG. **14**B. As shown, for each instruction and operation page, as indicated by the function block **144**, the system calculates the protection based on the new value for the PS field in step **146**. In step **148** the PowerPC page register tables are updated in step **148** with the new values for the protection attributes. Subsequently, the system flushes the PowerPC TLB entries in step **150** and returns to step **144** for processing another instruction or operand page register.

Power PC interrupts are handled by a Power PC Interrupt Service Routine **70**. These interrupts are passed to the software emulator and in particular an Interrupts 1750A object **71**. Interrupt handling does not form a part of the present invention.

Obviously, many modifications and variations of the present invention are possible in light of the above teachings. Thus, it is to be understood that, within the scope of the appended claims, the invention may be practiced otherwise than as specifically described above.

What is claimed and desired to be secured by Letters Patent of the United States is:

1. A memory system comprising:

one or more logical page registers, at least one of said logical page register configured to map 64K bytes of logical address space to sixteen (16) 4K word logical pages on 4K word boundaries; and

means for mapping each of said 4K word logical pages to four (4) 4K byte physical address pages, wherein each word in logical address space is mapped to 4 bytes in said physical address space; wherein said mapping means implemented in hardware.

2. The memory system as recited in claim **1**, further including means for providing one or more protection attributes for said memory.

3. The memory system as recited in claim **2**, wherein said protection attribute is block protection of 1K word blocks of physical memory space.

4. The memory system as recited in claim **2**, wherein said protection attribute is access lock and key.

5. The memory system as recited in claim **1**, wherein said mapping means includes a microprocessor conforming to the Power PC architecture.

6. A method for mapping logical address space to physical memory space, the method comprising the steps of:

(a) providing one or more logical page registers configured to map 64K words of logical address space to sixteen (16) 4K word logical pages on 4K word logical pages on 4K word boundaries; and

(b) mapping each of said 4K word logical pages to four (4) 4K byte physical address pages, wherein each word in logical address space is mapped to 4 bytes in said address pages.

7. The method as recited in claim **6**, further including the step of providing one or more protection attributes for said memory.

8. The method as recited in claim **7**, wherein said protection attribute is block protection of 1K word blocks in logical memory space.

9. The method as recited in claim **7**, wherein said protection attribute is access lock and key.

10. The method as recited in claim **6**, wherein said mapping is done in hardware.

11. The method as recited in claim **10**, wherein said hardware in a Power PC microprocessor.

* * * * *

US006467007B1

(12) **United States Patent**
Armstrong et al.

(10) **Patent No.: US 6,467,007 B1**
(45) **Date of Patent: Oct. 15, 2002**

(54) **PROCESSOR RESET GENERATED VIA MEMORY ACCESS INTERRUPT**

(75) Inventors: **Troy David Armstrong**, Rochester; **William Joseph Armstrong**, Kasson; **Naresh Nayar**, Rochester; **Kenneth Charles Vossen**, Kasson, all of MN (US)

(73) Assignee: **International Business Machines Corporation**, Armonk, NY (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/314,769**

(22) Filed: **May 19, 1999**

(51) **Int. Cl.**[7] **G06F 13/24**; G06F 15/177; G06F 11/30; G06F 1/24

(52) **U.S. Cl.** **710/260**; 710/266; 711/166; 713/1; 714/23; 714/763

(58) **Field of Search** 710/260, 261, 710/266; 709/215, 216; 712/216, 226, 244; 713/1, 100; 714/23, 701, 702, 763; 711/100, 147, 153, 166

(56) **References Cited**

U.S. PATENT DOCUMENTS

3,641,505 A	2/1972	Artz et al.	340/172.5
4,511,964 A	4/1985	Georg et al.	364/200

(List continued on next page.)

FOREIGN PATENT DOCUMENTS

WO	95/18998	* 7/1995	.....	G06F/1/24

OTHER PUBLICATIONS

Abstract for JAPIO Application No. 94–103092, T. Imada et al., Apr. 15, 1994, "Virtual Computer System."

Abstract for JAPIO Application No. 92–348434, T. Imada et al., Dec. 3, 1992, "Virtual Computer System."

Inspec Abstract No. C9408–6110P–022, A. B. Gargaro et al., Mar. 1994, "Supporting Distribution and Dynamic Reconfiguration in AdaPT."

VMWare Virtual Platform—Technology White Paper, http:vmware.com/products/virtualplatform.html (1999).

IBM Technical Disclosure Bulletin Kreulen, "OS/2 Raw FileSystem," vol. 40, No. 05, pp. 177–190, May 1997.

IBM Technical Disclosure Bulletin, Baskey et al., "Highly Parallel Coupling Facility Emulator/Router with Shadowed Link Buffers," vol. 39, No. 12, pp. 123–124 Dec. 1996.

U.S. Patent Application RO999–021, "Apparatus and Method for Specifying Maximum Interactive Performance in a Logical Partition of a Computer System Independently from the Maximum Interactive Performance in Other Partitions," filed May 19, 1999, Armstrong et al.

U.S. Patent Application RO999–023, "Management of a Concurrent Use License in a Logically–Partitioned Computer," filed May 19, 1999, Armstong et al.

U.S. Patent Application RO999–024, "Event–Driven Communications Interface for Logically–Partitioned Computer," filed May 19, 1999, Armstrong et al.

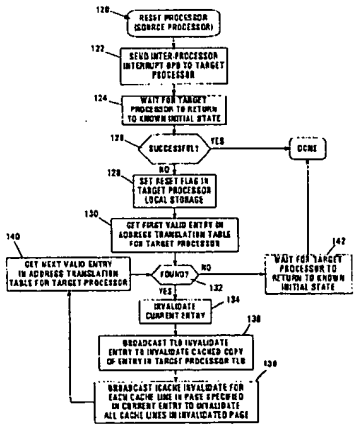
(List continued on next page.)

Primary Examiner—Xuan M. Thai
(74) *Attorney, Agent, or Firm*—Wood, Herron & Evans, LLP

(57) **ABSTRACT**

An apparatus, program product, and method utilize a memory access interrupt to effect a reset of a processor in a multi-processor environment. Specifically, a source processor is permitted to initiate a reset of a target processor simply by generating both a reset request and a memory access interrupt for the target processor. The target processor is then specifically configured to detect the presence of a pending reset request during handing of the memory access interrupt, such that the target processor will perform a reset operation responsive to detection of such a request.

33 Claims, 3 Drawing Sheets



U.S. PATENT DOCUMENTS

4,601,008	A	*	7/1986	Kato 710/260
4,843,541	A		6/1989	Bean et al. 364/200
4,924,378	A		5/1990	Hershey et al. 364/200
5,129,088	A		7/1992	Auslander et al. 395/700
5,204,897	A		4/1993	Wyman 380/4
5,253,344	A		10/1993	Bostick et al. 395/275
5,263,158	A		11/1993	Janis 395/600
5,297,287	A	*	3/1994	Miyayama et al. 713/1
5,345,590	A		9/1994	Ault et al. 395/650
5,365,514	A		11/1994	Hershey et al. 370/17
5,375,206	A		12/1994	Hunter et al.
5,446,902	A		8/1995	Islam 395/700
5,465,360	A	*	11/1995	Miller et al. 713/1
5,526,488	A		6/1996	Hershey et al. 395/200.2
5,550,970	A		8/1996	Cline et al. 395/161
5,566,337	A		10/1996	Szymanski et al. 395/733
5,574,914	A		11/1996	Hancock et al. 395/650
5,600,805	A		2/1997	Fredericks et al. 395/825
5,659,756	A		8/1997	Hefferon et al. 395/726
5,659,786	A		8/1997	George et al. 395/653
5,671,405	A		9/1997	Wu et al. 395/607
5,675,791	A		10/1997	Bhide et al. 395/621
5,684,974	A		11/1997	Onodera 395/412
5,687,363	A		11/1997	Oulid-Aissa et al. 395/604
5,692,174	A		11/1997	Birely et al. 395/603
5,692,182	A		11/1997	Desai et al. 395/610
5,742,757	A		4/1998	Hamadani et al. 395/186
5,784,625	A	*	7/1998	Walker 710/260
5,819,061	A		10/1998	Glassen et al. 395/406
5,828,882	A		10/1998	Hinckley 395/680
5,845,146	A		12/1998	Onodera 395/822
5,923,890	A		7/1999	Kubala et al. 395/800.01
5,948,065	A		9/1999	Eilert et al. 709/226
5,978,857	A		11/1999	Graham 709/301
5,996,026	A		11/1999	Onodera et al. 710/3
6,021,438	A		2/2000	Duvvoori et al. 709/224
6,061,695	A		5/2000	Slivka et al. 707/513
6,075,938	A		6/2000	Bugnion et al. 395/500.48
6,148,323	A		11/2000	Whitner et al. 709/105
6,173,337	B1		1/2001	Akhond et al. 709/318
6,199,179	B1		3/2001	Kauffman et al. 714/26
6,247,109	B1		6/2001	Kleinsorge et al. 712/13
6,263,359	B1		7/2001	Fong et al. 709/103
6,269,391	B1		7/2001	Gillespie 709/100
6,269,409	B1		7/2001	Solomon 709/329
6,282,560	B1		8/2001	Eilert et al. 709/100

OTHER PUBLICATIONS

U.S. Patent Application RO999–025, "Logical Partition Manager and Method," filed May 19, 1999, Armstrong et al.

Hauser, Ralf, "Does licensing require new access control techniques?", *Communications of the ACM,* vol. 37, No. 11, (Nov. 1994), pp. 48–55; Dialog copy pp. 1–10.

McGilton, Henry et al., *Introducing the UNIX System,* R.R. Donnelly & Sons Company, (1983), pp. 515–521.

Gomes, Lee, "Desktops to get OS freedom of choice," Wall Street Journal Online, Mar. 26, 1999.

VMWare 1.0x for Linux Changelog Archive (1999).

* cited by examiner

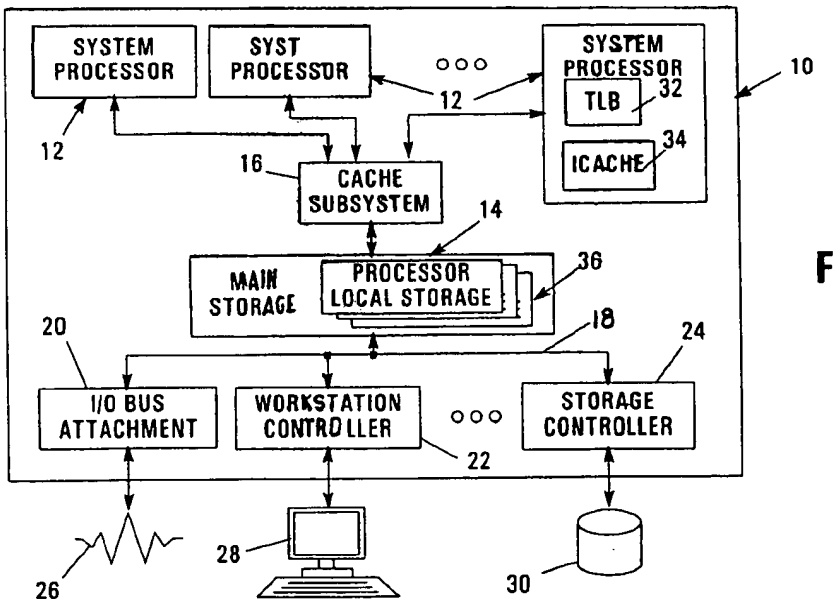

SYSTEM PROCESSOR
SYST PROCESSOR
SYSTEM PROCESSOR
TLB
32
ICACHE
34
10
12
16
CACHE SUBSYSTEM
14
MAIN STORAGE
PROCESSOR LOCAL STORAGE
36
18
20
24
I/O BUS ATTACHMENT
WORKSTATION CONTROLLER
STORAGE CONTROLLER
22
26
28
30

FIG. 1

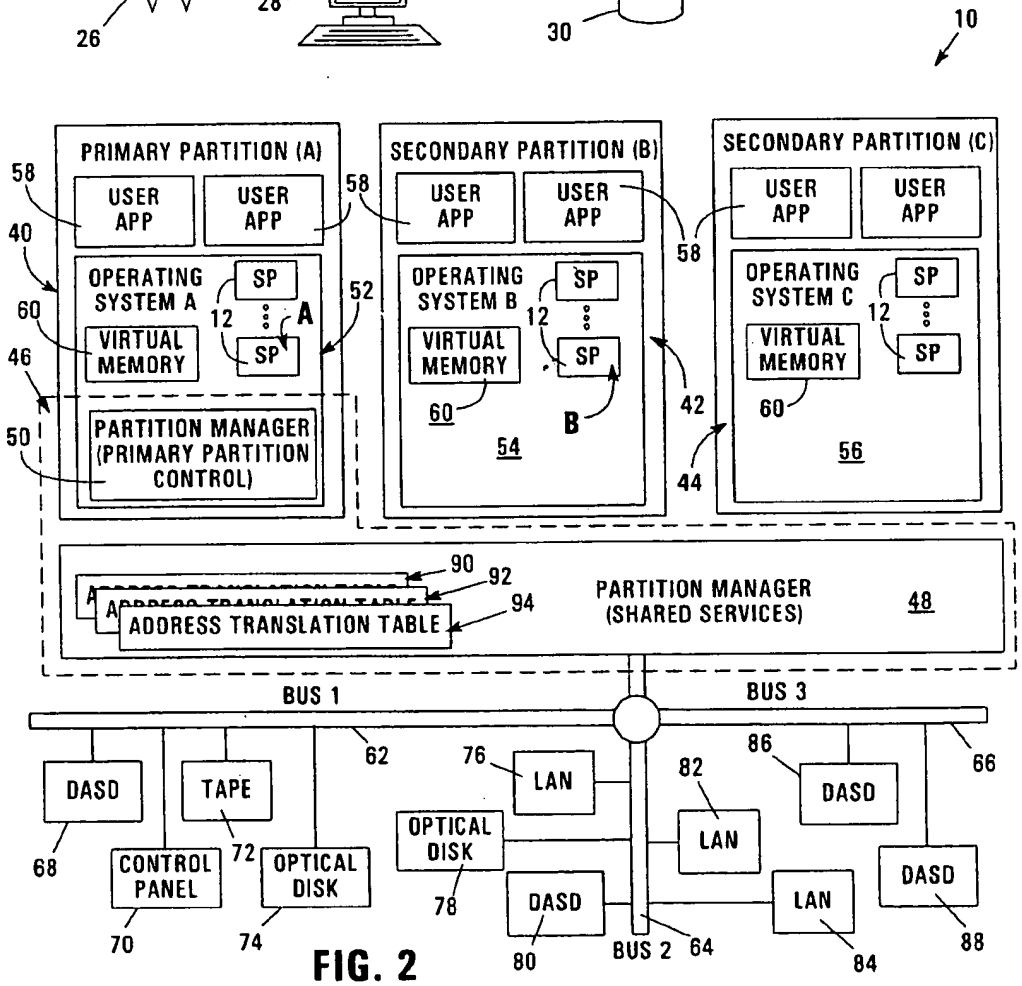

10
PRIMARY PARTITION (A)
SECONDARY PARTITION (B)
SECONDARY PARTITION (C)
USER APP
OPERATING SYSTEM A
OPERATING SYSTEM B
OPERATING SYSTEM C
SP
VIRTUAL MEMORY
PARTITION MANAGER (PRIMARY PARTITION CONTROL)
PARTITION MANAGER (SHARED SERVICES)
48
ADDRESS TRANSLATION TABLE
54
56
58
40
60
46
50
52
42
44
90
92
94
BUS 1
BUS 2
BUS 3
62
64
66
DASD
TAPE
CONTROL PANEL
OPTICAL DISK
LAN
68
70
72
74
76
78
80
82
84
86
88

FIG. 2

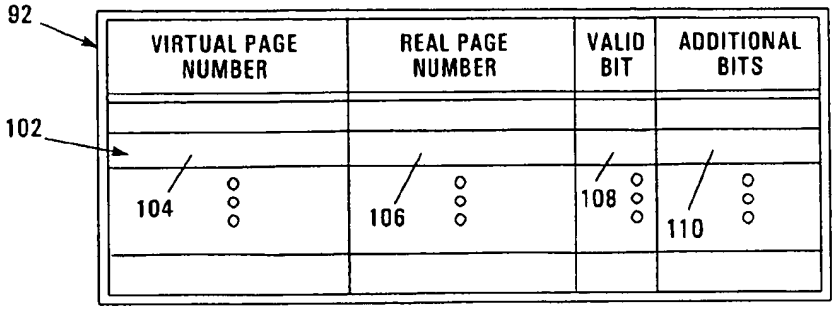

92
102
VIRTUAL PAGE NUMBER
REAL PAGE NUMBER
VALID BIT
ADDITIONAL BITS
104
106
108
110

FIG. 3

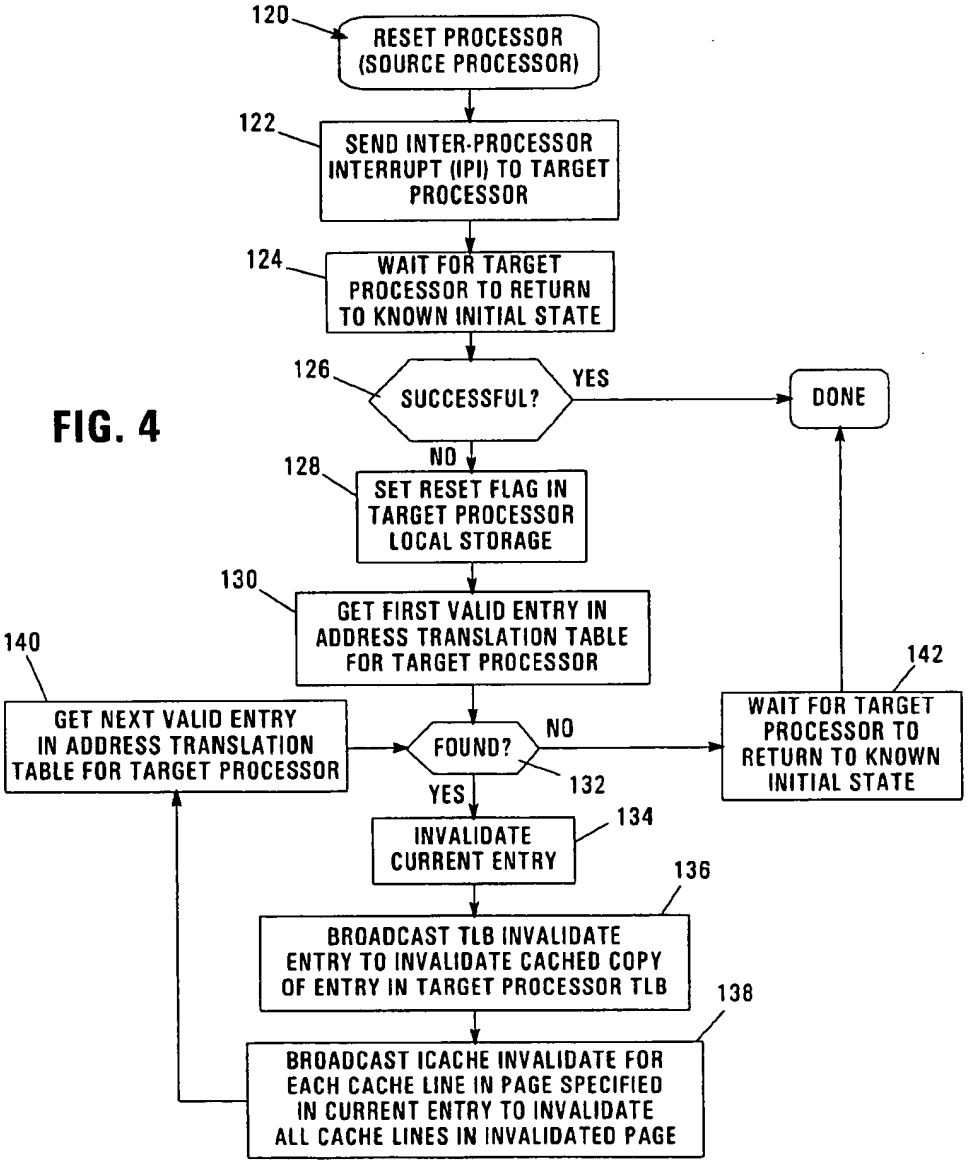

120
RESET PROCESSOR (SOURCE PROCESSOR)
122
SEND INTER-PROCESSOR INTERRUPT (IPI) TO TARGET PROCESSOR
124
WAIT FOR TARGET PROCESSOR TO RETURN TO KNOWN INITIAL STATE
126
SUCCESSFUL?
YES
DONE
NO
128
SET RESET FLAG IN TARGET PROCESSOR LOCAL STORAGE
130
GET FIRST VALID ENTRY IN ADDRESS TRANSLATION TABLE FOR TARGET PROCESSOR
140
GET NEXT VALID ENTRY IN ADDRESS TRANSLATION TABLE FOR TARGET PROCESSOR
FOUND?
NO
132
142
WAIT FOR TARGET PROCESSOR TO RETURN TO KNOWN INITIAL STATE
YES
134
INVALIDATE CURRENT ENTRY
136
BROADCAST TLB INVALIDATE ENTRY TO INVALIDATE CACHED COPY OF ENTRY IN TARGET PROCESSOR TLB
138
BROADCAST ICACHE INVALIDATE FOR EACH CACHE LINE IN PAGE SPECIFIED IN CURRENT ENTRY TO INVALIDATE ALL CACHE LINES IN INVALIDATED PAGE

FIG. 4

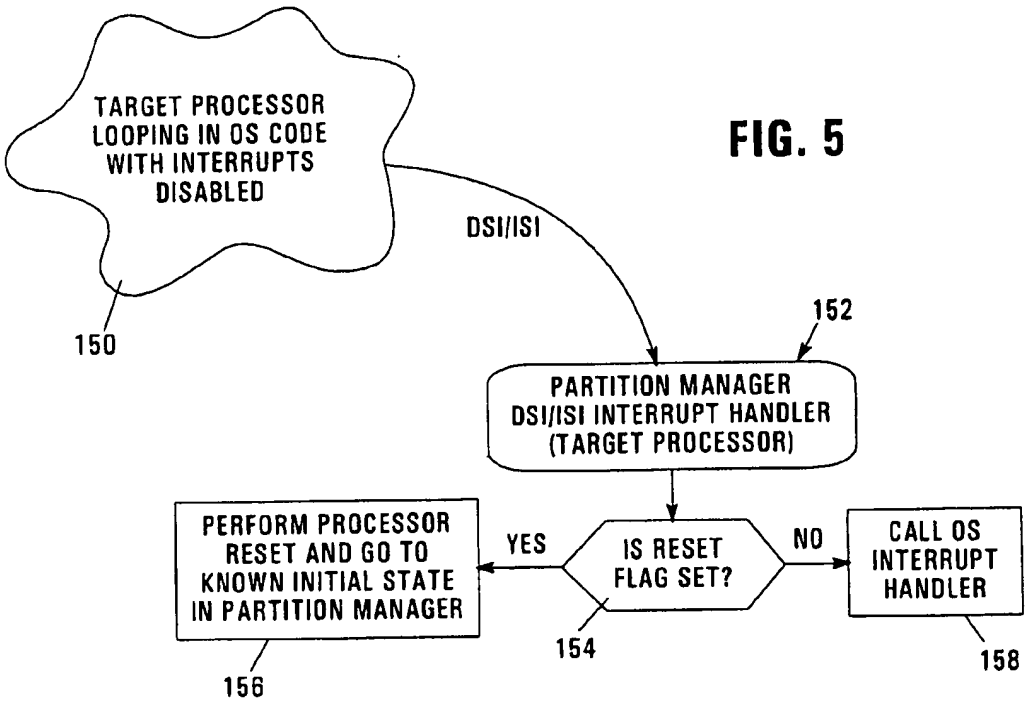

TARGET PROCESSOR LOOPING IN OS CODE WITH INTERRUPTS DISABLED
150
DSI/ISI
152
PARTITION MANAGER DSI/ISI INTERRUPT HANDLER (TARGET PROCESSOR)
IS RESET FLAG SET?
154
YES
NO
PERFORM PROCESSOR RESET AND GO TO KNOWN INITIAL STATE IN PARTITION MANAGER
156
CALL OS INTERRUPT HANDLER
158

FIG. 5

PROCESSOR RESET GENERATED VIA MEMORY ACCESS INTERRUPT

RELATED APPLICATIONS

This patent application is related to the following patent applications: U.S. patent application Ser. No. RO999-021, entitled "Apparatus and Method for Specifying Maximum Interactive Performance in a Logical Partition of a Computer System Independently from the Maximum Interactive Performance in Other Partitions," Ser. No. 09/314,541 filed May 19, 1999 by Armstrong et al.; U.S. patent application Ser. No. RO999-023, entitled "Management of a Concurrent Use License in a Logically-Partitioned Computer," Ser. No. 09/314,324 filed May 19, 1999 by Armstrong et al.; U.S. patent application Ser. No. RO999-024, entitled "Event-Driven Communications Interface for Logically-Partitioned Computer," Ser. No. 09/4314,187 filed May 19, 1999 by Armstrong et al.; and U.S. patent application Ser. No. RO999-025, entitled "Logical Partition Manager and Method," Ser. No. 09/314,214 filed May 19, 1999 by Armstrong et al.

FIELD OF THE INVENTION

The invention is generally related to computers and computer software. In particular, the invention is generally related to initiating a reset of a computer processor via a software-based mechanism.

BACKGROUND OF THE INVENTION

Computer technology continues to advance at a rapid pace, with significant developments being made in both software and in the underlying hardware upon which such software executes. One significant advance in computer technology is the development of multi-processor computers, where multiple computer processors are interfaced with one another to permit multiple operations to be performed concurrently, thus improving the overall performance of such computers. Also, a number of multi-processor computer designs rely on logical partitioning to allocate computer resources to further enhance the performance of multiple concurrent tasks.

With logical partitioning, a single physical computer is permitted to operate essentially like multiple and independent "virtual" computers (referred to as logical partitions), with the various resources in the physical computer (e.g., processors, memory, input/output devices) allocated among the various logical partitions. Each logical partition executes a separate operating system, and from the perspective of users and of the software executing on the logical partition, operates as a fully independent computer.

A shared resource, often referred to as a "hypervisor" or partition manager, manages the logical partitions and facilitates the allocation of resources to different logical partitions. As a component of this function, a partition manager maintains separate virtual memory address spaces for the various logical partitions so that the memory utilized by each logical partition is fully independent of the other logical partitions. One or more address translation tables are typically used by a partition manager to map addresses from each virtual address space to different addresses in the physical, or real, address space of the computer. Then, whenever a logical partition attempts to access a particular virtual address, the partition manager translates the virtual address to a real address so that the shared memory can be accessed directly by the logical partition.

A primary benefit of multi-processor computers, and in particular of those implementing partitioned environments, is the ability to maintain at least partial operational capability in response to partial system failures. For example, while most computers, and in particular most multi-processor computers, are relatively reliable, the processors in such computers can "hang" from time to time and cease to operate in responsive and predictable manners, e.g., due to software design flaws, or "bugs", that cause such processors to operate continuously in endless loops. In a partitioned environment in particular, hanging a processor allocated to a particular logical partition often results in that partition becoming at least partially inoperative and non-responsive. However, other logical partitions that do not rely on the hung processor are typically not affected by the failure.

While it may be acceptable in some situations to permit a computer to simply be powered off and on to recover from a hung processor, in many situations it is more desirable to provide the ability for a hung processor to be reset, or restored to a known state, in such a manner that the entire computer does not need to be shut down. Also, in a multi-processor computer, and in particular one that implements a partitioned environment, it is often desirable for such a reset operation to not affect other processors and/or other logical partitions operating in the computer so that the other processors and/or logical partitions can still perform useful operations while the hung processor is reset.

In many multi-processor computers, and in particular in those implementing partitioned environments, a software-based reset mechanism is typically supported to permit one processor to initiate a reset of another processor. Typically, a software-based reset mechanism relies on the use of interrupts, often referred to as inter-processor interrupts (IPI's), to cause a hung processor to reset and restore itself to a known state. An IPI, like all interrupts, causes a processor to cease all current operations and immediately jump to dedicated program code, referred to as an "interrupt handler", to handle the interrupt.

An IPI is typically handled as an "external" interrupt insofar as an IPI is initiated externally from the processor that receives the interrupt. Most processors, however, support the ability to selectively enable or disable external interrupts so that such interrupts will be ignored—typically when a processor is executing relatively critical program code that should not be terminated prior to completion. The ability to disable external interrupts, however, introduces the possibility that a processor may hang while external interrupts are disabled, and thus be incapable of being reset through an IPI. Should this occur, the only manner of resetting the processor would likely be a hardware reset, which would typically necessitate a full restart of the computer, and a consequent temporary inaccessibility of the computer.

Therefore, a significant need exists for an alternate software-based reset mechanism for a processor that permits the processor to be reset in wider range of situations, and in particular, for a software-based reset mechanism for a processor that cannot be defeated as a result of the disabling of interrupts on the processor.

SUMMARY OF THE INVENTION

The invention addresses these and other problems associated with the prior art by providing an apparatus, program product, and method that utilize a memory access interrupt to effect a reset of a processor in a multi-processor environ-

ment. Specifically, one processor (referred to herein as a source processor) is permitted to initiate a reset of another processor (referred to herein as a target processor) simply by generating both a reset request and a memory access interrupt for the target processor. The target processor is then specifically configured to detect the presence of a pending reset request during handing of the memory access interrupt, such that the target processor will perform a reset operation responsive to detection of such a request.

Detection of a reset request is typically implemented within an interrupt handler that is executed by a target processor in response to a memory access interrupt. As a result, for those situations in which a memory access interrupt is generated for a reason other than to initiate a reset of the target processor, the target processor can handle the interrupt in an appropriate manner, and often with little additional overhead associated with determining whether a reset operation should be performed as a result of the interrupt.

A memory access interrupt may be considered to include any type of interrupt that is generated responsive to a memory access attempt by the target processor. Particularly given the general necessity for a processor to always be capable of accessing memory, a memory access interrupt is often further characterized as being incapable of being disabled during the operation of the target processor. As a consequence, unlike external interrupts such as IPI's and the like which are capable of being disabled in some instances, a reset operation can be initiated on a target processor via a memory access interrupt irrespective of whether other interrupts are disabled on the processor.

While other alternative memory access interrupt implementations may also be utilized consistent with the invention, one particularly useful implementation relies on a type of memory access interrupt that is generated in response to an attempt by a target processor to access a virtual memory address in a virtual memory address space that is not mapped by any entry in an address translation table. Generation of a memory access interrupt then typically requires only that one or more entries in the address translation table be invalidated to ensure that a subsequent access to the virtual memory address space will attempt to access an unmapped virtual memory address.

Therefore, consistent with one aspect of the invention, a processor may be reset by generating a reset request for the processor, generating a memory access interrupt on the processor, and resetting the processor during handling of the memory access interrupt by the processor responsive to detection of the reset request.

These and other advantages and features, which characterize the invention, are set forth in the claims annexed hereto and forming a further part hereof. However, for a better understanding of the invention, and of the advantages and objectives attained through its use, reference should be made to the Drawings, and to the accompanying descriptive matter, in which there is described exemplary embodiments of the invention.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a computer consistent with the invention.

FIG. 2 is a block diagram of the primary software components and resources in the computer of FIG. 1.

FIG. 3 is a block diagram of an address translation table in FIG. 2.

FIG. 4 is a flowchart illustrating the program flow of a reset processor routine executed by a source processor in the computer of FIGS. 1 and 2.

FIG. 5 is a flowchart illustrating the program flow of a partition manager interrupt handler executed by a target processor in the computer of FIGS. 1 and 2, in response to a memory access interrupt.

DETAILED DESCRIPTION

Hardware and Software Environment

Turning to the Drawings, wherein like numbers denote like parts throughout the several views, FIG. 1 illustrates a data processing apparatus or computer 10 consistent with the invention. Apparatus 10 generically represents, for example, any of a number of multi-user computer systems such as a network server, a midrange computer, a mainframe computer, etc. However, it should be appreciated that the invention may be implemented in other data processing apparatus, e.g., in stand-alone or single-user computer systems such as workstations, desktop computers, portable computers, and the like, or in other computing devices such as embedded controllers and the like. One suitable implementation of apparatus 10 is in a midrange computer such as the AS/400 series computer available from International Business Machines Corporation.

Apparatus 10 generally includes a plurality of system processors 12 coupled to a memory subsystem including main storage 14, e.g., an array of dynamic random access memory (DRAM). System processors 12 may be configured, for example, to implement a symmetric multiprocessing (SMP) environment, among other possible multi-processor environments. Also illustrated as interposed between processors 12 and main storage 14 is a cache subsystem 16, typically including one or more levels of data, instruction and/or combination caches, with certain caches either serving individual processors or multiple processors as is well known in the art. Furthermore, main storage 14 is coupled to a number of types of external (I/O) devices via a system bus 18 and a plurality of interface devices, e.g., an input/output bus attachment interface 20, a workstation controller 22 and a storage controller 24, which respectively provide external access to one or more external networks 26, one or more workstations 28, and/or one or more storage devices such as a direct access storage device (DASD) 30.

In the illustrated embodiment, computer 10 is implemented as a logically partitioned computer. In this regard, FIG. 2 illustrates in greater detail the primary software components and resources utilized in implementing a logically partitioned computing environment on computer 10, including a plurality of logical partitions 40, 42, 44 managed by a partition manager 46. Any number of logical partitions may be supported as is well known in the art. Moreover, it will be appreciated that the invention may be utilized in other partitioned environments, as well as in other computers (e.g., non-partitioned computers) that utilize multiple processors.

In the illustrated implementation, logical partition 40 operates as a primary partition, while logical partitions 42 and 44 operate as secondary partitions. A primary partition in this context shares some of the partition management functions for the computer, such as handling the powering on or powering off of the secondary logical partitions on computer 10, or initiating a memory dump of the secondary logical partitions. As such, a portion of partition manager 46 is illustrated by primary partition control block 50, disposed in the operating system 52 resident in primary partition 40. Other partition management services, which are accessible by all logical partitions, are represented by shared services

block **48**. Implementation of partition management functionality within a primary logical partition is described, for example, in U.S. patent application Ser. No. RO999-025, entitled "Logical Partition Manager and Method, Ser. No. 09/314,214, filed on May 19, 1999 by Armstrong et al. , which is incorporated by reference herein. However, partition management functionality need not be implemented within any particular logical partition in other implementations consistent with the invention.

Each logical partition utilizes an operating system, e.g., operating systems **52**, **54** and **56** for logical partitions **40**, **42** and **44**, respectively), that controls the primary operations of the logical partition in the same manner as the operating system of a non-partitioned computer. For example, each operating system **52–56** may be implemented using the OS/400 operating system available from International Business Machines Corporation, among others, residing on top of a kernel, e.g., AS/400 system licensed internal code (SLIC). The shared services in block **48** are alternatively referred to herein as partition licensed internal code (PLIC). Also illustrated are several user applications **58** that execute on each logical partition **40–44** and rely on the underlying services provided by the operating systems thereof.

Each logical partition **40–44** executes in a separate memory space, represented by virtual memory **60**. Moreover, each logical partition **40–44** is statically and/or dynamically allocated a portion of the available resources in computer **10**. For example, each logical partition is allocated one or more processors **12**, as well as a portion of the available memory space for use in virtual memory **60**. Logical partitions can share specific hardware resources such as processors, such that a given processor is utilized by more than one logical partition. In the alternative hardware resources can be allocated to only one logical partition at a time.

Additional resources, e.g., mass storage, backup storage, user input, network connections, and the like, are typically allocated to one or more logical partitions in a manner well known in the art. Resources can be allocated in a number of manners, e.g., on a bus-by-bus basis, or on a resource-by-resource basis, with multiple logical partitions sharing resources on the same bus. Some resources may even be allocated to multiple logical partitions at a time. FIG. **2** illustrates, for example, three logical buses **62**, **64** and **66**, with a plurality of resources on bus **62**, including a direct access storage device (DASD) **68**, a control panel **70**, a tape drive **72** and an optical disk drive **74**, allocated to primary logical partition **40**. Bus **64**, on the other hand, may have resources allocated on a resource-by-resource basis, e.g., with local area network (LAN) adaptor **76**, optical disk drive **78** and DASD **80** allocated to secondary logical partition **42**, and LAN adaptors **82** and **84** allocated to secondary logical partition **44**. Bus **66** may represent, for example, a bus allocated specifically to logical partition **44**, such that all resources on the bus, e.g., DASD's **86** and **88**, are allocated to the same logical partition.

It will be appreciated that the illustration of specific resources in FIG. **2** is merely exemplary in nature, and that any combination and arrangement of resources may be allocated to any logical partition in the alternative. Moreover, it will be appreciated that in some implementations resources can be reallocated on a dynamic basis to service the needs of other logical partitions. Furthermore, it will be appreciated that resources may also be represented in terms of the input/output processors (IOP's) used to interface the computer with the specific hardware devices.

The various software components and resources illustrated in FIG. **2** and implementing the embodiments of the invention may be implemented in a number of manners, including using various computer software applications, routines, components, programs, objects, modules, data structures, etc., referred to hereinafter as "computer programs", or simply "programs". The computer programs typically comprise one or more instructions that are resident at various times in various memory and storage devices in the computer, and that, when read and executed by one or more processors in the computer, cause that computer to perform the steps necessary to execute steps or elements embodying the various aspects of the invention. Moreover, while the invention has and hereinafter will be described in the context of fully functioning computers, those skilled in the art will appreciate that the various embodiments of the invention are capable of being distributed as a program product in a variety of forms, and that the invention applies equally regardless of the particular type of signal bearing medium used to actually carry out the distribution. Examples of signal bearing media include but are not limited to recordable type media such as volatile and non-volatile memory devices, floppy and other removable disks, hard disk drives, magnetic tape, optical disks (e.g., CD-ROM's, DVD's, etc.), among others, and transmission type media such as digital and analog communication links.

In addition, various programs described hereinafter may be identified based upon the application for which they are implemented in a specific embodiment of the invention. However, it should be appreciated that any particular program nomenclature that follows is used merely for convenience, and thus the invention should not be limited to use solely in any specific application identified and/or implied by such nomenclature.

Those skilled in the art will recognize that the exemplary environments illustrated in FIGS. **1** and **2** are not intended to limit the present invention. Indeed, those skilled in the art will recognize that other alternative hardware and/or software environments may be used without departing from the scope of the invention.

Processor Reset Via Memory Access Interrupt

The embodiments described hereinafter generally operate by utilizing the existing memory access interrupt handling functionality of a processor to initiate a reset of the processor. A memory access interrupt can generally be considered to include any type of interrupt that is generated responsive to a memory access attempt by a processor. For example, in the implementation discussed hereinafter, a memory access interrupt is implemented as either or both of a data storage interrupt and instruction storage interrupt that is generated whenever a processor attempts to access a virtual memory address in a virtual memory address space that is not mapped to any real memory address by an address translation table utilized by that processor. However, it will be appreciated that memory access interrupts may be generated in other manners consistent with the invention, e.g., in response to a miss of a translation lookaside buffer, among others.

A number of existing hardware and software components in computer **10** are specifically utilized when initiating a processor reset in the manner disclosed herein. For example, as shown in FIG. **2**, one or more address translation tables, e.g., address translation tables **90**, **92** and **94** (also referred to as hardware page tables (HPT's)) are provided in partition manager **46** to respectively handle the virtual to real address translation operations for logical partitions **40**, **42** and **44**, respectively. Moreover, as shown in FIG. **1**, each processor **12** optionally includes a translation lookaside buffer (TLB)

32 or other cache structure that caches at least a portion of one or more address translation tables to accelerate the translation of virtual to real memory addresses, in a manner well known in the art.

Each processor **12** typically also includes one or more levels of instruction cache, e.g., level one instruction cache (ICache) **34**, within which one or more cache lines are stored. Other relevant cache structures may also be found in cache subsystem **16**. Furthermore, as illustrated at **36**, at least a portion of main storage is allocated for local storage for one or more processors, which local storage is statically allocated to a fixed region of memory addresses to permit persistent access to the local storage at all times.

To initiate a processor reset in the manner described herein, a first processor (referred to herein as a "source processor"), which desires to initiate a processor reset of another processor (referred to herein as a "target processor"), typically must generate a reset request and a memory access interrupt for the target processor. In FIG. **2**, an exemplary source processor allocated to primary logical partition **40** is illustrated at "A", and an exemplary target processor allocated to secondary logical partition **42** is illustrated at "B". However, it should be appreciated that source and target processors may be allocated to the same logical partition in some implementations.

In the illustrated embodiment, generation of a reset request is implemented via setting a flag located at a static memory location in the local storage for the target processor. The reset flag may alternatively be implemented in any other memory storage device that is accessible (at least indirectly) to both the source and target processors. Moreover, other manners of generating a reset request or otherwise indicating to a target processor that a reset is requested will be appreciated by one of ordinary skill in the art having the benefit of the instant disclosure.

Also in the illustrated embodiment, generation of a memory access interrupt is implemented by invalidating, with the source processor, every entry in an address translation table associated with the target processor. Doing so ensures that the next time the target processor attempts to access any memory address (be it to retrieve a next instruction or to access data stored in memory), a memory access interrupt will be generated. In addition, to maintain coherency, it is desirable to update any caching mechanisms (such as TLB's) to invalidate any cached entries from an invalidated address translation table, and/or to update any other caching mechanisms that cache data and/or instructions associated with any such invalidated entries.

In other embodiments, it may not be necessary to invalidate every entry of an address translation table. For example, it may be possible in some embodiments to invalidate entries one at a time until a memory access interrupt is detected by the source processor. It may also be possible in some implementations to predict which entry will be accessed next, and only invalidate that entry. Furthermore, in other embodiments different caching mechanisms may or may not need to be updated to maintain coherency.

In addition, other mechanisms for generating a memory access interrupt may be utilized consistent with the invention. For example, as discussed above, a memory access interrupt may be initiated by generating a miss on a translation lookaside buffer or other address translation data caching structure.

Other alternatives will be apparent to one of ordinary skill in the art.

FIG. **3** illustrates in greater detail a suitable implementation of address translation table **92** allocated to logical partition **42** and used by target processor B (FIG. **2**). As is well known in the art, an address translation table includes a plurality of entries, e.g., entry **102**, including a plurality of fields **104**, **106**, **108** and **110**. Address translation in the illustrated embodiment occurs on a page-by-page basis, e.g., with a page size of 4096 bytes. Each entry **102** thus matches a page of virtual memory address to a corresponding page of real memory addresses in the memory system. The "page" of a memory address is typically identified by those bits from the memory address other than the lowest order number of bits corresponding to the page size. Thus, for a page size of 4096, as well as a 64-bit memory address space, a page is identified by the upper 42 bits (bits **0–41**, where bit **0** is the MSB), with the low order 12 bits (bits **42–63**) utilized to specify a particular memory address in an identified page. As such, in the illustrated implementation, field **104** of each entry **102** includes a 42-bit virtual page number, with entry **106** including a 42-bit real page number to which the virtual page is mapped. It should be appreciated that either or both of the virtual and real memory address spaces may have differing sizes consistent with the invention.

Each entry **102** further includes a valid field **108** storing a bit that indicates whether or not the entry represents a valid mapping of a virtual page to a real page. It is this bit that is cleared by a source processor whenever it is desired to generate a memory access interrupt on a target processor that utilizes address translation table **92**.

Additional information, represented by field **110**, may also be stored within an entry **102** in an address translation table **92**. Typically, such additional information includes various protection bits, as well as reference, change, address compare and/or other information known in the art. It should be appreciated that other data structures may be utilized in an address translation scheme consistent with the invention.

FIG. **4** next illustrates a reset processor routine **120**, executed by a source processor whenever it is desirable to reset a target processor to an initial state. Routine **120** may be called, for example, whenever it is detected that a processor, or a logical partition associated with that processor, has become non-responsive (here, after an unsuccessful IPI). Other situations in which it may be desirable to reset a processor include power off, main store dump, and continuously powered mainstore (CPM) initial program load (IPL) of a secondary partition, among others.

First, as shown at block **122**, the source processor may attempt to send an inter-processor interrupt (IPI) to the target processor, in a manner known in the art. Next, the processor waits at block **124** for the target processor to return to a known initial state, e.g., by setting a timer and periodically checking the responsiveness of the target processor. In such an implementation, expiration of the timer without a response from the target processor would indicate an unsuccessful reset operation.

Next, in block **126**, it is determined whether the target processor has returned to its known initial state. If so, routine **120** is complete. If not, however, control passes to **128** to perform memory access-based processor reset consistent with the invention.

Specifically, in block **128**, the source processor sets a reset flag in the local storage for the target processor to be reset. Next, in block **130**, the source processor attempts to retrieve the first valid entry in the address translation table allocated to the target processor. Assuming such an entry is found, block **132** passes control to block **134** to invalidate the current entry, typically by clearing the valid bit therefor. Next, in block **136**, any copy of the entry in the translation

lookaside buffer (TLB) for the target processor is invalidated by broadcasting a TLB invalidate entry message, the use and configuration of which is well understood in the art.

Next, in block **138**, all of the cache lines in the page that is being invalidated are invalidated in the target processor's instruction cache by broadcasting an instruction cache block invalidate (ICBI) instruction for each cache line referenced in the page. The use and operation of an ICBI instruction are well understood in the art.

Blocks **136** and **138** essentially maintain coherency between the address translation table and any cached copies of any entries referenced thereby, as well as any cached copies of cache lines incorporated within any invalidated pages. It should be appreciated, however, that in other embodiments, coherency issues may not be present, and either or both of blocks **136** or **138** may be omitted.

After broadcast of the ICBI instructions, control passes to block **140** to attempt to obtain the next valid entry in the address translation table associated with the target processor. Control then returns to block **132** to determine whether another such entry was found. Processing then continues until each valid entry in the address translation table has been invalidated. Once all such entries have been processed, block **132** passes control to block **142** to wait for the target processor to return to its known initial state. After this occurs, routine **120** is complete.

Implementation of the functionality of routine **120** is typically within program code allocated to partition manager **46** (FIG. **2**), principally within shared services block **48**. Specifically, block **50** in the primary logical partition portion of the partition manager determines whenever a processor needs to be reset, and initiates routine **120** in shared services block **48** to implement such a reset. Other allocations of functionality between blocks **48** and **50** may be used in the alternative, however.

It should be appreciated that additional modifications may be made to routine **120** consistent with the invention. For example, rather than searching for only valid entries, all entries of an address translation table may be processed in the manner disclosed herein. Moreover, it may not be necessary or desirable in some embodiments to attempt an IPI prior to a memory access-based interrupt. Moreover, waiting for the target processor to return to its known initial state, as disclosed in connection with blocks **124** and **142** may be performed in a number of alternate manners, including setting a watchdog timer, etc. Other modifications will be apparent to one of ordinary skill in the art.

FIG. **5** next illustrates the initiation of a processor reset by the target processor in response to the generation of a memory access interrupt by the source processor (described above in connection with FIG. **4**). As shown at **150**, the target processor is illustrated as looping in operating system code with its interrupts disabled. In response to a data storage interrupt or instruction storage interrupt, control passes to a partition manager DSI/ISI interrupt handler **152** executed by the target processor. Routine **152** is initiated any time a memory access interrupt is generated on the target processor, irrespective of whether the memory access interrupt was generated in response to a request to reset the processor. As such, routine **152** begins in block **154** by determining whether the reset flag allocated to the target processor in the local storage therefor is set. If so, control passes to block **156** to perform a processor reset and go to a known initial state in the partition manager code, in a manner well understood in the art. By returning to such a known initial state, the reset is achieved, in a manner specifically adapted for the particular configuration of computer **10**.

Returning to block **154**, if the reset flag is not set, the memory access interrupt is handled in a conventional manner, e.g., by passing control to block **158** to call an interrupt handler in the operating system code, as is also well known in the art. Typically, handling of such an interrupt includes determining whether the requested virtual address is valid, and if so, retrieving a real address from the operating system's software page table. Then, the retrieved real address is stored in the address translation table in the partition manager, and normal processing is resumed. If the virtual address is not a valid address for the logical partition, an exception is created, which is handled in the operating system in a manner understood in the art.

Various additional modifications may be made consistent with the invention. For example, other manners of insuring coherency between an address translation table and other components in the computer may be used in the alternative.

Other modifications may be made to the illustrated embodiments without departing from the spirit and scope of the invention. Therefore, the invention lies in the claims hereinafter appended.

What is claimed is:

1. A method of resetting a processor, comprising:

(a) generating a reset request for the processor;

(b) generating a memory access interrupt on the processor; and

(c) resetting the processor during handling of the memory access interrupt by the processor responsive to detection of the reset request.

2. The method of claim **1**, further comprising accessing data with the processor using a virtual memory system accessible through an address translation table associated with the processor, and wherein generating the memory access interrupt includes invalidating at least one entry in the address translation table.

3. The method of claim **2**, wherein invalidating at least one entry in the address translation table includes invalidating every entry in the address translation table.

4. The method of claim **3**, wherein generating the memory access interrupt further includes invalidating a corresponding entry in a translation table cache associated with the processor.

5. The method of claim **4**, wherein generating the memory access interrupt further includes invalidating a cache entry, associated with a cache line referenced by an invalidated entry in the address translation table, and located in a cache associated with the processor.

6. The method of claim **1**, wherein the address translation table includes a plurality of entries, with each entry identifying a virtual memory address used by the processor and a real memory address mapped to the virtual memory address.

7. The method of claim **1**, wherein generating the reset request includes setting a reset flag.

8. The method of claim **7**, wherein the reset flag is stored in a local storage area for the processor.

9. The method of claim **7**, further comprising detecting the reset request in the processor by accessing the reset flag.

10. The method of claim **1**, wherein the memory access interrupt includes at least one of a data storage interrupt and an instruction storage interrupt.

11. The method of claim **1**, wherein resetting the processor during handling of the memory access interrupt responsive to detection of the reset request is performed by a memory access interrupt routine executed by the processor responsive to a memory access interrupt.

12. The method of claim **11**, wherein generating the reset request and generating the memory access interrupt are performed on a second processor coupled to the first processor.

13. The method of claim **12**, wherein the first and second processors are among a plurality of processors in a multiprocessor computer system.

14. The method of claim 13, wherein the multi-processor computer system defines a plurality of partitions, each partition including an operating system executing on at least one of the plurality of processors, the multi-processor computer system further including a partition manager executing on at least one of the plurality of processors, wherein generating the reset request and generating the memory access interrupt are performed by the partition manager.

15. The method of claim 1, further comprising detecting whether the processor may be locked up with external interrupts disabled, wherein generating the reset request and generating the memory access interrupt are performed responsive to detecting that the processor may be locked up with external interrupts disabled.

16. The method of claim 1, wherein generating the reset request and generating the memory access interrupt are performed responsive to determining that an attempt to reset the processor using an inter-processor interrupt was unsuccessful after determining that the processor is locked up.

17. A method of resetting a first processor among a plurality of processors in a multi-processor computer system, comprising:

(a) detecting with a processor other than the first processor a likely lockup condition in the first processor while external interrupts on the first processor are disabled;

(b) generating with a processor other than the first processor a reset request for the first processor;

(c) generating with a processor other than the first processor a memory access interrupt on the first processor; and

(d) handling the memory access interrupt in the first processor, including detecting the reset request with the first processor and performing a reset on the first processor responsive thereto.

18. An apparatus comprising first and second processors, wherein:

(a) the second processor is configured to initiate a reset of the first processor by generating a reset request and a memory access interrupt for the first processor; and

(b) the first processor is configured to handle the memory access interrupt and to perform a reset responsive to detection of the reset request during handling of the memory access interrupt.

19. The apparatus of claim 18, wherein the first processor is further configured to access data from a memory using an address translation table associated with the first processor, and wherein the second processor is configured to generate the memory access interrupt by invalidating at least one entry in the address translation table.

20. The apparatus of claim 19, the second processor is configured to generate the memory access interrupt by invalidating every entry in the address translation table.

21. The apparatus of claim 20, further comprising a translation table cache associated with the first processor, wherein the second processor is further configured to invalidate a corresponding entry in the translation table cache.

22. The apparatus of claim 21, further comprising a cache associated with the first processor, wherein the second processor is further configured to invalidate a cache entry in the cache that is associated with a cache line referenced by an invalidated entry in the address translation table.

23. The apparatus of claim 22, wherein the cache is a level one instruction cache.

24. The apparatus of claim 18, wherein the second processor is configured to generate the reset request by setting a reset flag, and wherein the first processor is configured to detect the reset request by determining whether the reset flag is set.

25. The apparatus of claim 18, wherein the memory access interrupt includes at least one of a data storage interrupt and an instruction storage interrupt.

26. The apparatus of claim 18, wherein the first processor is configured to execute a memory access interrupt routine responsive to the memory access interrupt, and to detect the reset request during execution of the memory access interrupt routine.

27. The apparatus of claim 18, wherein the first and second processors are among a plurality of processors in a multi-processor computer system, wherein the multi-processor computer system defines a plurality of partitions, each partition including an operating system executing on at least one of the plurality of processors, the multi-processor computer system further including a partition manager executing on at least the second processor.

28. The apparatus of claim 18, wherein the second processor is configured to detect whether the first processor may be locked up with external interrupts disabled, and to generate the reset request and the memory access interrupt responsive to detecting that the first processor may be locked up with external interrupts disabled.

29. The apparatus of claim 18, wherein the second processor is configured to generate the reset request and the memory access interrupt responsive to determining that an attempt to reset the first processor using an inter-processor interrupt was unsuccessful.

30. An apparatus, comprising:

(a) a memory defining a real address space with a plurality of real memory addresses;

(b) a first processor configured to access the memory using a virtual address space including a plurality of virtual memory addresses;

(c) an address translation table accessible by the first processor and including a plurality of entries, each entry configured to map a virtual memory address in the virtual address space to a real memory address in the real address space;

(d) a memory access interrupt handler configured to be executed by the first processor in response to an attempt by the first processor to access an unmapped virtual memory address in the address translation table, the memory access interrupt handler further configured to reset the first processor responsive to a pending reset request; and

(e) a second processor coupled to the first processor, the second processor configured to initiate a reset of the first processor by generating a reset request and invalidating at least one entry in the address translation table.

31. The apparatus of claim 30, wherein the plurality of processors are configured to implement a partitioned environment including a plurality of logical partitions.

32. A program product, comprising:

(a) first and second programs respectively configured to execute on first and second processors, the second program configured to initiate a reset of the first processor by generating a reset request and a memory access interrupt for the first processor, and the first program configured to handle the memory access interrupt and to perform a reset responsive to detection of the reset request during handling of the memory access interrupt; and

(b) a signal bearing medium bearing the first and second programs.

33. The program product of claim 32, wherein the signal bearing medium includes at least one of a recordable medium and a transmission-type medium.

* * * * *